ISBN 978-82-471-1558-9 (printed ver.)
ISBN 978-82-471-1560-2 (electronic ver.)
ISSN 1503-8181

NTNU
Norwegian University of
Science and Technology

Anita Gupta

Doctoral Thesis

Doctoral theses at NTNU, 2009:90

NTNU
Norwegian University of
Science and Technology
Thesis for the degree of
philosophiae doctor
Faculty of Information Technology, Mathematics and
Electrical Engineering
Department of Computer and Information Science

NTNU

Doctoral theses at NTNU, 2009:90

Anita Gupta

The Profile of Software Changes in Reused vs. Non-Reused Industrial Software Systems

NTNU
Norwegian University of
Science and Technology

Anita Gupta

The Profile of Software Changes in Reused vs. Non-Reused Industrial Software Systems

Thesis for the degree of philosophiae doctor

Trondheim, May 2009

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

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of philosophiae doctor

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

©Anita Gupta

ISBN 978-82-471-1558-9 (printed ver.)
ISBN 978-82-471-1560-2 (electronic ver.)
ISSN 1503-8181

Doctoral Theses at NTNU, 2009:90

Printed by Tapir Uttrykk

Abstract

High-quality software, delivered on time and budget, constitutes a critical part of most products and services in modern society. Our ability to develop and maintain such software is still inadequate. However, software reuse and Component-Based Software Engineering (CBSE) seem to be the potential technologies to reduce time-to-market, and achieve better software quality.

Development *for* reuse refers to the deliberate development of software components that can be reused. Development *with* reuse refers to the inclusion of these reusable components in new software. Since the 1970s work has been ongoing to study the issues related to software reuse, maintenance and evolution. The focus has been on how to develop for/with reuse, technical/managerial/organizational aspects, assessing the effect of reuse in terms of defect density and change density, and reporting the success and failure of reuse practices.

The research in this thesis is based on several empirical studies performed in a large Norwegian oil and gas company (StatoilHydro ASA in Stavanger/Trondheim, Norway), and in the context of the SEVO (Software Evolution in Component-Based-Software Engineering) project, financed in part by the Research Council of Norway. Data have been collected for all releases of a reused class framework, called Java Enterprise Framework (JEF), as well as from two applications reusing the framework "as-is", namely Digital Cargo Files (DCF) and Shipment and Allocation (S&A). The main focus has been to investigate the relation between software changes and software reuse, and propose reuse guidelines based on these insights. The research has followed a combined quantitative and qualitative design approach. Quantitative data were collected from the company's data repositories for three releases of a reused class framework, and for the two applications. The qualitative data were collected by interviewing senior developers, and reading numerous documents, web pages and other studies.

The following research questions have been identified in this thesis:

RQ1: *What is the relation between software changes and software reuse, by comparing the reused framework vs. software reusing it?*

RQ2: *How do the reused framework and software reusing it evolve over time?*

RQ3: *What improvements can be made towards the actual reuse practice at StatoilHydro ASA?*

The main contributions are:

C1: *Identification of differences/similarities of the change profile for the reused framework vs. software reusing it.*

C2: *Identification of differences/similarities of the defect profile for the reused framework vs. software reusing it.*

C3: *Description of the software change lifecycle for the reused framework vs. software reusing it.*

C4: *Identification of possible software reuse improvements.*

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) as partial fulfilment of the requirements for the degree of Philosophiae Doctor.

This doctoral work has been performed at the Department of Computer and Information Science, NTNU, Trondheim, under the supervision of Professor Reidar Conradi. Professor Tor Stålhane and Professor Eric Monteiro have been the co-advisors.

This work was conducted as a part of the SEVO (Software Evolution in Component-Based Software Engineering) research project, supported by the Research Council of Norway through contract number 159916/V30. StatoilHydro ASA cooperated with the SEVO project. StatoilHydro ASA is a merger of Statoil ASA and Hydro, effective from fall 2007. We used Statoil ASA before the merger and StatoilHydro ASA after the merger.

In this thesis I have generally used “we” to present the work, describing both my own reflections on the work in Chapter 1-7 and the collaborative studies described in the papers P1 to P6 in Appendix A. Research is a collaborative process and I have received valuable feedback from many colleagues, especially from my supervisor.

Except for minor formatting adjustments, all of the papers from P1 to P6 in Appendix A have been presented in their original version.

75% of the PhD scholarship was funded through the SEVO project, and the remaining 25% was funded by one year of compulsory teaching assistant duties at the Department of Computer and Information Science, NTNU. The scholarship lasted from 2004-2008.

Acknowledgements

This thesis is part of the SEVO project (Software Evolution in Component-Based Software Engineering) at the Norwegian University of Science and Technology (NTNU). The doctoral work was financed by SEVO through the Research Council of Norway for three years, and I gratefully acknowledge this support. I also wish to thank NTNU for helping to fund the work through my duties as a research assistant, which lasted for one year. The fieldwork was done at StatoilHydro ASA in Trondheim/Stavanger, Norway.

First of all I would like to thank my supervisor Professor Reidar Conradi for his continuous support and advice during my PhD study, and his comments on the papers and drafts of this thesis. I also thank other members of the SEVO 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 Dr. Jingyue Li and Professor Tor Stålhane.

I would also like to thank StatoilHydro ASA, especially Harald Rønneberg and Einar Landre, for involving me in their projects, and giving me the opportunity to perform empirical studies and share the results. I am deeply grateful for this opportunity and want to thank everyone at StatoilHydro ASA who helped me.

Finally, I would like to thank my family, for their love and support.

NTNU, December 11, 2008
Anita Gupta

Contents

List of Figures

List of Tables

Abbreviations

AOP	Aspect-Oriented Programming
CASE	Computer-Aided Software Engineering
CBD	Component-Based Development
CBSE	Component-Based Software Engineering
CCB	Change Control Board
COM+	Component Object Model
CORBA	Common Object Request Broker Architecture
CR	Change Request
COTS	Commercial-Off-The-Shelf
DCF	Digital Cargo File
EJB	Enterprise JavaBeans
GQM	Goal Question Metric
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
IT	Information Technology
JEF	Java Enterprise Framework
J2EE	Java 2nd Enterprise Edition
KNSLOC	Kilo Non-commented Source Line of Code
MDA	Model Driven Architecture
.NET	NETwork
NSLOC	Non-commented Source Line of Code
NTNU	Norwegian University of Science and Technology
O&S	Oil Sales, Trading & Supply
ODC	Orthogonal Defect Classification
OSS	Open Source Software
PDM	Physical Deal Maintenance
QA	Quality Assurance
RC	Research Challenge
RCA	Root Cause Analysis
RQ	Research Question
S&A	Shipment & Allocation
SCM	Software Configuration Management
SEI	Software Engineering Institute
SEVO	Software EVOlution
SLOC	Source Line of Code
SPORT	StatoilHydro's Planning and Operational System for Offshore, Refinery and Terminals
TR	Trouble Report

1 Introduction

This chapter describes the background (Section 1.1) and the context for the research (Section 1.2). In addition, this chapter also presents research questions (Section 1.3), research design (Section 1.4), the list of papers (Section 1.5), the contributions (Section 1.6) and finally the thesis structure (Section 1.7).

1.1 Problem Outline

High-quality software, delivered on time and budget, constitutes a critical part of most products and services in modern society. Our ability to develop and use components (see Section 2.4 for definition) as building blocks has significantly enhanced system development. Components are more coarse-grained than objects, which might be beneficial in retrieving and assembly, and since they conform to a component model (see Section 2.4 for definition) they also facilitate composition. This is why the commercial sector and academia have both shown interest in component-based software development, and much effort has been devoted to define and describe the terms and concepts involved.

Even though component-based system development offers potential benefits, such as improved quality and productivity, reduced time-to-market, reduced cost and a commodity-oriented perspective of software, our ability to develop and maintain such software is still inadequate. This is partly due to the massive and unexpected software evolution (accumulated changes) during the development and maintenance of software, both regarding products and related processes.

Prior studies [Lientz78] [Pigoski97] [Bennett00] [Mohagheghi04a] have investigated several aspects of maintenance. Examples include the variations in the amount of maintenance activities, in which part of the development process these activities are located, and what the consequences of these activities are. Due to the dynamic nature of software, we need to revisit these questions and answers to ensure that the findings remain valid. This may also make it possible to discover new or additional results. Some of the main challenges are to investigate how software evolution affects the changes to a software system, and how to manage this evolution. In order to handle these issues, developers' knowledge and experience need to be taken into account, and combined with the empirical findings from the software code itself.

Records of software changes are an important source of information for studying software reuse, evolution and maintenance. Such changes are frequent in most software systems, and are responsible for a large part of the software costs. Empirical studies of industrial software systems may help us to understand the volume and nature of software evolution, and they may answer questions about how organizations can manage their processes and resources.

This thesis will investigate the software change profile in terms of defect profile and change profile. The former will consider defect types, the severities of defects and the impact of defects[1]. The latter will consider the classification of changes in maintenance activities. It will investigate change trends related to issues such as priority levels and component size in the reused framework and software reusing it, in a large Norwegian oil and gas company, StatoilHydro ASA.

1.2 Research Context

The research in this thesis relies on quantitative and qualitative empirical studies of all releases of a reused class framework, as well as from two applications reusing the framework "as-is", developed by StatoilHydro ASA in Trondheim/Stavanger, Norway.

The company initiated its reuse strategy in 2003 with pre-studies. A central IT strategy of the O&S (Oil Sales, Trading and Supply) business area has been to explore the potential benefits of reusing software systematically. This strategy was started as a response to the changing business and market trends, in order to provide a consistent and resilient technical platform for development and integration. The company has invested in development *for* reuse (i.e., the deliberate development of software components that can be reused), and also in development *with* reuse (i.e., the inclusion of these reusable components into new software) [Sindre95]. The Java framework for developing Enterprise Applications is called the JEF framework or *JEF* in this thesis. Components in the framework are hereafter termed *reused components* (reused in two distinct applications). Components in the *application* are specific to each application and are termed *non-reused components*. This means that the components in the applications were not initially designed and implemented for future reuse, like JEF. To decide what components to include in JEF was based on a simple rule from the management in the company: *If a developed application produces a functionality that can be useful for other applications, this functionality will be made generic and included in the JEF framework.* The project leader of JEF is the one who decides what functionalities should be made generic and included as a component in JEF. There are several documentations and tools related to the reuse practice in StatoilHydro ASA. Examples include several JEF documents and one wiki, documentations about Rational ClearCase and ClearQuest, etc., and all these documents have been read by the developers working with reuse projects. JEF consists of seven separate components or classes, which can be reused separately or together, and they are built and reused in-house. JEF is reused "as-is" in two applications, namely Digital Cargo Files (DCF,

[1] Impact refers to what the user has or would have noticed if the defect persists after application deployment at the user's site.

which is the abbreviation used throughout this thesis) and Shipment and Allocation (S&A, which is the abbreviation used throughout this thesis). This means that all the code of the reused components was included in each application and was thus reused "as-is". *DCF* is mainly a document storage application to manage cargo files. A "cargo file" is a container for working documents that are related to a contract or cargo that is used by all parties in the company. *S&A* is an application for doing efficient and controllable business processes through common business principles within lift and cargo planning.

Figure 1 gives an overview of software change data collected in the company, and how software changes are defined in this context.

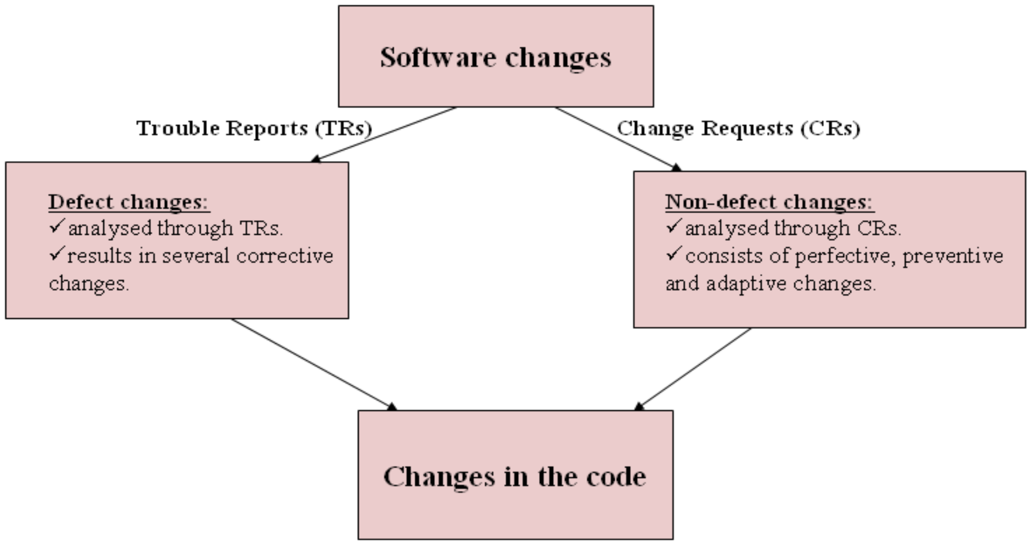

Figure 1. Software change definition

According to Figure 1, software changes can be classified as *defect* or *non-defect* changes. Defects can be analysed through Trouble Reports (TRs) and non-defect changes through Change Requests (CRs), but both TRs and CRs can be rejected, redefined or postponed, and both can lead to changes in project reports and documents. In addition to analysing TRs and CRs, the present work has also collected data of changes made to the source code, and the analysed data is reported in papers **P4** and **P6**.

StatoilHydro ASA has supported this work in collecting data and performing studies. The thesis work is done in the context of the SEVO (Software Evolution in Component-Based Software Engineering) project, which is a Norwegian research project (contract number 159916/V30) from 2004-2008. SEVO defines the following four main project goals:

G1) Better understanding of software evolution, focusing on CBSE technology.
G2) Better methods to predict the risks, costs, and profile of software evolution in CBSE.
G3) Contributing to national competence based around these themes.

G4) Dissemination and exchange of the knowledge gained.

1.3 Research Questions

The overall research goal (**RG**) for all studies carried out as part of this thesis was:

- *Investigate the advantages/disadvantages of systematic software reuse and the reasons behind it, by analysing software change data (including both defect and non-defect changes). Then, based on these insights, propose specific reuse guidelines (as an example of improvements) to StatoilHydro ASA, as well as general recommendations to software practitioners.*

In order to go from our overall goal to specific studies and research questions, we have formulated the following questions:

What are we studying?

- We performed several empirical studies (i.e., survey and case studies) in a large Norwegian oil and gas company. The objective for this research was to investigate the relation between software changes and software reuse and propose reuse guidelines based on these insights. *Software changes* refer here to all kinds of changes done on software systems, i.e., defect changes and non-defect changes (see Figure 1).

Why are we interested in it?

- Studying defect changes increases our understanding of the relationship between systematic reuse and the defect density of the reused software, and reveals several decision-making factors that pertain to that relationship.
- Studying non-defect changes increases our understanding as to whether reused software and its applications follow similar or different change profiles over time.

Why would this be of any interest to anyone else?

- This benefits both the research community and practitioners. The former gains deeper insight into benefits and challenges of software reuse with respect to evolution and maintenance. The latter gain insight into how to implement more systematic reuse policies to help reduce the defect density and change density of the software change data of the software developed for reuse.

The thesis presents six studies, and we have formulated three research questions that explore our overall goal. We present the research questions briefly in this section, and give a rationale for why these questions are seen as important in Chapter 4.

RQ1: What is the relation between software changes and software reuse, by comparing the reused framework vs. software reusing it?
RQ2: How do the reused framework and software reusing it evolve over time?
RQ3: What improvements can be made towards the actual reuse practice at StatoilHydro ASA?

1.4 Research Design

Empirical studies can be qualitative, quantitative, or a combination of both. The choice of approach affects the data collection, data analysis and discussions of validity. The studies performed in this thesis have been a combination of qualitative and quantitative approaches. The first phase in Figure 2 consists of a combined quantitative and qualitative study, where a survey of industrial software reuse has been done. The second phase consists of mainly quantitative studies on CRs and TRs. The third phase consists also mainly of quantitative studies of software change data related to the source code. A combination of quantitative and qualitative research methods in this thesis has several purposes [Mohagheghi04b]:

- Performing studies that are both exploratory and confirmatory.
- Expanding our understanding and confirming results of one study by other studies.
- Exploiting all available data collected; both quantitative data such as software change data (i.e., CRs, TRs and change data related to the source code), and qualitative data such as interviews with developers, project reports and process descriptions.

Figure 2 shows the type of studies performed, the year and sequence, as well as the relation to papers and contributions. The papers numbered from **P1** to **P6** are listed in Section 1.5, and the contributions numbered from **C1** to **C4** are described in Section 1.6.

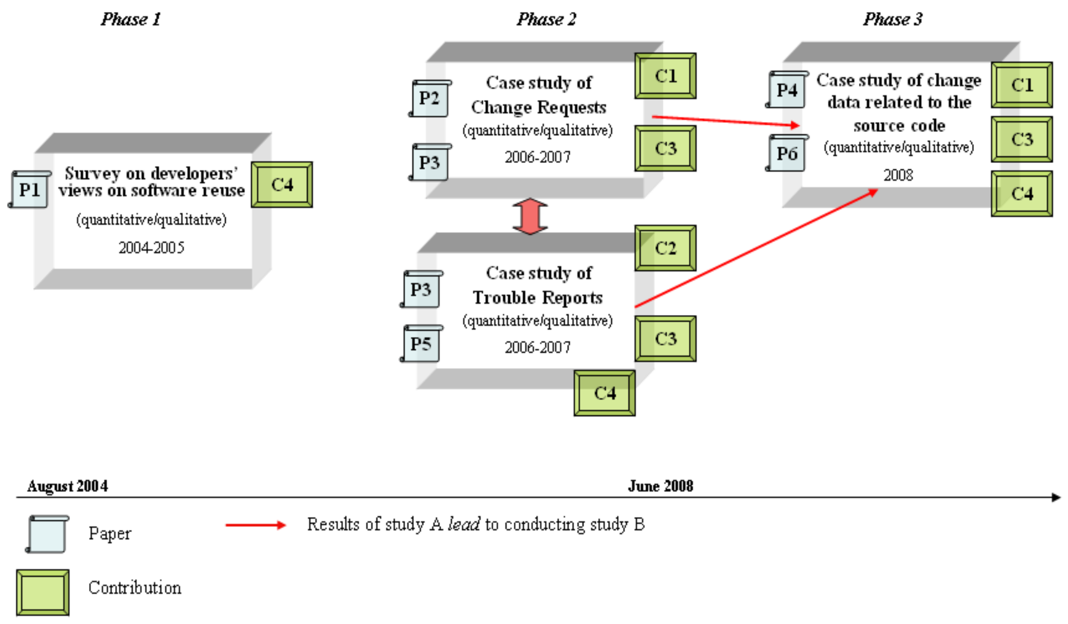
Phase 1
Phase 2
Phase 3
P1
Survey on developers' views on software reuse
(quantitative/qualitative)
2004-2005
C4
P2
P3
Case study of Change Requests
(quantitative/qualitative)
2006-2007
C1
C3
P3
P5
Case study of Trouble Reports
(quantitative/qualitative)
2006-2007
C2
C3
C4
P4
P6
Case study of change data related to the source code
(quantitative/qualitative)
2008
C1
C3
C4
August 2004
June 2008
Paper
Contribution
Results of study A lead to conducting study B

Figure 2. The studies with their related papers and contributions

Figure 2 outlines four main studies, and Table 1 provides an overview of the studies, their focus and research methods. The context has been StatoilHydro ASA in all four studies, hence "context" is not included in the table.

Table 1. Overview of the studies

Study	Year	Focus	Research method
Study 1	2004-2005	Study of developers' views on software reuse	Survey followed by semi-structured interviews
Study 2	2006	Analysing Trouble Reports (TRs)	Case study
Study 3	2007	Analysing Change Requests (CRs)	Case study
Study 4	2008	Analysing change data related to the source code	Case study

The research methods for each research question are:

- **RQ1 and RQ2**: will be investigated by *data mining, quantitative analysis* and in some cases a *qualitative Root Cause Analysis* (*RCA*). *The investigations will be on* CRs, TRs and software change data related to the source code stored in two different configuration management tools, the company's internal measures and documents, as well as *qualitative* interviews with senior developers.
- **RQ3**: will be answered by *combining results* from **RQ1** and **RQ2.** This will be done in two ways. Firstly, by proposing improvements in the overall cause-effect relationship model between systematic software reuse and the possible lower defect density of the reused software. Secondly, by gaining insight into how the software lifecycle for the reused framework and software reusing it evolves over time. Additionally, **RQ3** will also be answered by conducting a survey followed by semi-structured interviews and both *qualitative* and *quantitative* knowledge will be gained on possible software reuse improvements.

1.5 Papers

This section gives a short summary of the 6 papers numbered **P1** to **P6** included in this thesis. Together they constitute the four main studies the results are based on. We briefly describe their relevance to this thesis and my contribution. I have stated the amount of my contribution in percentages, and with a small description of what my main work has been. The full papers are included in Appendix A. In addition, we have included abstracts of two other papers produced during the work on this thesis in Appendix B. The papers included in this thesis are marked P#, while the one secondary paper which is not included is marked SP#.

P1. Odd Petter N. Slyngstad, **Anita Gupta,** Reidar Conradi, Parastoo Mohagheghi, Harald Rønneberg, and Einar Landre: "An Empirical Study of Developers' Views on Software Reuse in Statoil ASA", In Jose Carlos Maldonado and Claes Wohlin (Eds.): Proc. 5th International Symposium on Empirical Software Engineering (ISESE'06), 21-22 September 2006, Rio de Janeiro, Brazil. IEEE CS Press, ISBN 1-59593-218-6, pp. 242-251.
Relevance to this thesis: This paper presents the results from a survey followed by semi-structured interviews, investigating opinions of developers on software reuse, and the results can help StatoilHydro ASA to improve their reuse practice.
My contribution: I was one of the leading authors in addition to Odd Petter Slyngstad and contributed 50% of the work, including research design, collecting and analysing change requests, and paper writing. The whole paper writing process started with Odd Petter Slyngstad and I dividing the work

among us. We then worked on the tasks separately, and when we were done we read through each other's contributions to make major or minor comments. We both made the questions in the questionnaire presented in the paper. However, after the questionnaire was filled out we performed separate follow-up interviews with the participants. In the end we wrote the discussion and conclusion part.

P2. **Anita Gupta**, Odd Petter N. Slyngstad, Reidar Conradi, Parastoo Mohagheghi, Harald Rønneberg, and Einar Landre: "An Empirical Study of Software Changes in Statoil ASA - Origin, Priority Level and Relation to Component Size", In Proc. International Conference on Software Engineering Advances (ICSEA'06), 29 October - 3 November 2006, Tahiti, French Polynesia. IEEE CS Press, ISBN 0-7695-2703-5, pp. 12- 19.
Relevance to this thesis: This paper describes the results of analysing change requests from four releases of a set of components developed for reuse. The results characterize and explain the changes to the components.
My contribution: I was one of the leading authors in addition to Odd Petter Slyngstad and contributed 50% of the work, including research design, collecting and analysing change requests, and paper writing. The whole paper writing process started with Odd Petter Slyngstad and I dividing the work among us. We then worked on the tasks separately, and when we were done we read through each other's contributions to make major or minor comments. We then classified all the change requests separately and cross-validated the results. In the end we wrote the discussion and conclusion part.

P3. **Anita Gupta**, Odd Petter N. Slyngstad, Reidar Conradi, Parastoo Mohagheghi, Harald Rønneberg, and Einar Landre: "A Case Study of Defect-Density and Change-Density and their Progress over Time", In René L. Krikhaar, Chris Verhoef, and Giuseppe A. Di Lucca (Eds.): Proc. 11th European Conference on Software Maintenance and Reengineering, Software Evolution in Complex Software Intensive Systems (CSMR'07), 21-23 March 2007, Amsterdam, The Netherlands. IEEE Computer Society, ISBN 0-7695-2802-3, pp. 7-16.
Relevance to this thesis: This paper explored the defect density and change request density and their progress over time for a reused framework, compared with one application that is reusing it. The results contribute towards understanding the maintenance and evolution of the reused framework and the software reusing it.
My contribution: I was one of the leading authors in addition to Odd Petter Slyngstad and contributed 50% of the work, including research design, collecting and analysing change requests and trouble reports, and paper writing. The whole paper writing process started with Odd Petter Slyngstad and I dividing the work among us. We then worked on the tasks separately, and when we were done we read through each other's contributions to make major or minor comments. We then analysed all the trouble reports and change requests separately and then cross-validated the results. In the end we wrote the discussion and conclusion part.

P4. **Anita Gupta**, Forrest Shull, Daniela Cruzes, Chris Ackermann, Reidar Conradi, Harald Rønneberg and Einar Landre: "Experience Report on the Effect of Software Development Characteristics on Change Distribution", In Andreas Jedlitschka and Outi Salo (Eds.): Proc. 9th International Conf. on Product Focused Software Development and Process Improvement (PROFES'08), 23-25 June 2008, Rome, Italy. Springer Verlag, ISBN 978-3-540-69564-6, pp. 158-173. The paper received the Best Paper Award at the conference (out of 31 papers), and was invited and later accepted for a special issue of Software Process Improvement and Practice (SPIP) journal.
Relevance to this thesis: This paper classified and compared software changes (related to the non-commented source code) for the reused class framework and one application reusing it. The results deepened our insight into the impact software changes have on different development characteristics (e.g. impact of reuse and impact of refactoring).
My contribution: I was the leading author and contributed 80% of the work, including research design, data collection, data analysis and paper writing. The whole paper writing process started with me dividing the work among the co-authors. I did most of the paper writing, data collection and analysis. The co-authors helped out with the data analysis, gave me their major or minor feedback, and did a proofreading of the final paper.

P5. **Anita Gupta**, Jingyue Li, Reidar Conradi, Harald Rønneberg and Einar Landre: "A Case Study Comparing Defect Profiles of a Reused Framework and of Applications Reusing it", accepted with minor revisions (2 May 2008) to the Journal of Empirical Software Engineering.
Relevance to this thesis: This paper has analysed trouble reports for a reused framework and two applications that are reusing it. The results of this study increases our understanding of the overall cause-effect relationship between systematic reuse and the possible lower defect density of the reused software, and reveals several decision-making factors that pertain to that relationship.
My contribution: I was the leading author and contributed 80% of the work, including research design, collecting and analysing trouble reports, and paper writing. The whole paper writing process was divided mostly between Dr. Jingyue Li and myself. I divided the work among us, and did most of the paper writing, data collection and analysis. However, Dr. Jingyue Li helped out with the data analysis (i.e., classified all the defects separately and then cross-validated the results) and paper writing (e.g. related work and discussion).

P6. **Anita Gupta**, Jingyue Li, Reidar Conradi, Harald Rønneberg and Einar Landre: "Change Profiles of a Reused Class Framework vs. two of its Applications", is submitted to Journal of Information and Software Technology (01.07.2008).
Relevance to this thesis: This paper has analysed change data related to the non-commented source code for a reused framework and two applications that are reusing it. The results of this study advance our understanding of factors that affect the change density and change profile for the reused framework vs. software reusing it.

My contribution: I was the leading author and contributed 80% of the work, including research design, data collection, data analysis, and paper writing. The whole paper writing process was divided mostly between Dr. Jingyue Li and myself. I divided the work among us, and did most of the paper writing, data collection and analysis. However, Dr. Jingyue Li helped out with the data analysis (i.e., he helped to classify those changes I was uncertain about) and paper writing (e.g. related work and discussion). The remaining co-authors gave me their feedback on the paper.

The paper designated as SP1 is considered to be outside the scope of this thesis. The paper designated as SP2 is related to the topic of this thesis, but overlaps with paper **P5**, already used in this thesis. Hence, paper SP2 is not included in Appendix A. We will not discuss the relevance of SP1 or SP2 to this thesis, or my contribution. The abstracts of these papers can be found in Appendix B.

SP1. **Anita Gupta,** Marianne H. Asperheim, Odd Petter N. Slyngstad, and Harald Rønneberg: "An Empirical Study of Distributed Technologies Used in Collaborative Tasks at Statoil ASA", In Enrico Blanzieri and Tao Zhang (Eds.): Proc. 2nd International Conference on Collaborative Computing (CollaborateCom'06), 17-20 November 2006, Atlanta, Georgia. IEEE CS Press, ISBN 1-4244-0429-0, pp. 1-5.

SP2. Jingyue Li, **Anita Gupta**, Jon Arvid Børretzen, and Reidar Conradi: "The Empirical Studies on Quality Benefits of Reusing Software Components", In Xiaodong Liu et al. (Eds.): Proc. 1st International Workshop on Quality Oriented Reuse of Software (QUORS'07), 23-27 July 2007, Beijing, China. IEEE CS Press, ISBN 978-0-7695-2870-0, pp. 399-402.

1.6 Contributions

This thesis has four main contributions:

C1 Identification of differences/similarities of the change profile for the reused framework vs. software reusing it.

- Using a quantitative analysis of Change Requests (CRs) for the reused framework (JEF), insight has been gained into the distribution of CRs, the data trend for how customers and developers assign priority to CRs, and issues regarding maintainability of large components in JEF.

- Using a quantitative analysis with a qualitative RCA of software changes related to the non-commented SLOC for the reused framework (JEF) and software reusing it (DCF), insight has been gained into the distribution of software changes for JEF vs. DCF. It was found that *designing for reuse* does have an effect on the change types (e.g. amount of changes and localization of changes), and a positive effect was seen for the *refactoring* that occurred in DCF.

- Using a quantitative analysis of software change data related to non-commented SLOC together with a qualitative RCA, the differences/similarities in the maintenance activities were explained for the reused framework vs. software reusing it by classifying changes into perfective, corrective, preventive and adaptive changes.

C2 Identification of differences/similarities of the defect profile for the reused framework vs. software reusing it.

- Using a quantitative analysis of Trouble Reports (TRs) together with a qualitative RCA for the reused framework vs. software reusing it, deeper insight was gained into the overall cause-effect relationship between systematic reuse and the possible lower defect density of the reused software, and several decision-making factors that pertain to that relationship were revealed.

C3 Description of the software change lifecycle for the reused framework vs. software reusing it.

- Using a quantitative analysis of Trouble Reports (TRs) and Change Requests (CRs) for the reused framework and software reusing it, it was found how defect density and change request density progress over time.

- Using a quantitative analysis of software change data related to non-commented SLOC together with a qualitative RCA, it was found how the reused framework vs. software reusing it evolves over time, according to the Bennett and Rajlich stage model for describing the lifecycle of a software system.

C4 Identification of possible software reuse improvements.

- Using a quantitative and qualitative analysis of a survey that was conducted, and by combining the results from **RQ1** and **RQ2**, possible improvements were made towards the software reuse practice by exploring the benefits/factors contributing to better software reuse. This made an improved overall cause-effect relationship model between systematic software reuse and the possible lower defect density of the reused software. In addition, it identified possible factors that affect the change density and change profile of reused software.

Table 2 shows the relation between research questions, papers and contributions.

Table 2. Research questions vs. contributions and papers

Research Question	Contribution	Papers	Focus
RQ1	C1,C2	P2, P4, P5, P6	Software changes in the context of software reuse.
RQ2	C3	P3, P6	Software evolution and maintenance over time in the context of software reuse.
RQ3	C4	P1, P5, P6	Software reuse improvements.

1.7 Thesis Structure

Figure 3 illustrates the structure of this thesis.

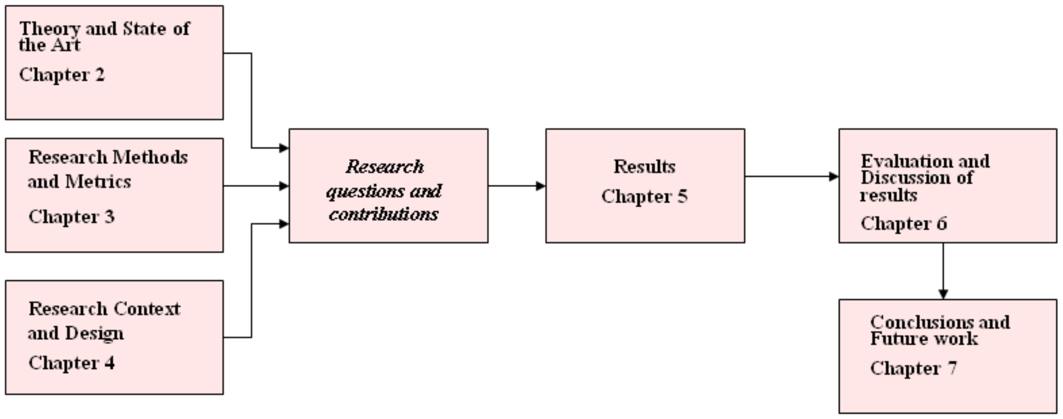
Theory and State of the Art
Chapter 2
Research Methods and Metrics
Chapter 3
Research Context and Design
Chapter 4
Research questions and contributions
Results
Chapter 5
Evaluation and Discussion of results
Chapter 6
Conclusions and Future work
Chapter 7

Figure 3. The structure of this thesis

Chapter 2: Briefly presents the field of software engineering and its state of the art, including software quality, software reuse, CBSE, and software evolution and maintenance. It also gives an overview of common research methods and metrics in software engineering, and a more detailed description of the research methods used in this thesis. Finally, the challenges faced in the context of comparing the profile of software changes for a reused framework and software reusing it are presented.

Chapter 3: Presents an overview of research methods and metrics, and challenges in selecting research methods.

Chapter 4: Introduces the research focus and the research context, describing the StatoilHydro ASA as a company, and the research design in this thesis.

Chapter 5: Presents the main results of the work. All the contributions from papers P1 to P6 are presented here.

Chapter 6: The research questions are further discussed in this chapter. A discussion and evaluation of the contributions and the results are made. The relations between research questions, papers, contributions and SEVO goals are presented. Finally, some reflections on the research context are discussed.

Chapter 7: Sums up the main findings from the discussion, and proposes future work.

Appendix A: Includes the six papers that have been submitted or published which form the basis for this thesis.

Appendix B: Presents the abstract of the two papers that were excluded from the final thesis.

2 State of the Art

This chapter present the rationale, definition, concepts and challenges of software engineering (Section 2.1), software quality (Section 2.2) and software reuse (Section 2.3). Next, Component-Based Software Engineering, CBSE (Section 2.4) and software evolution and maintenance (Section 2.5) are presented. Finally, in Section 2.6, the research challenges related to state of the art are presented.

2.1 Software Engineering: Definition and Challenges

Finkelstein and Kramer [Finkelstein00] define *software engineering* as:

… the branch of systems engineering concerned with the development of large and complex software intensive systems.

Additionally, it is also concerned with the technology (i.e., processes, methods, techniques and tools) for the development of software intensive systems within the budget and on schedule. Throughout this thesis, *technology* will be used as a common term to incorporate processes, methods, techniques, tools and related concepts, formalisms and languages. Software engineering activities or phases include managing, estimation, planning, modelling, analysing, specifying, designing, implementing, testing and maintaining [Fenton97].

Kruchten [Kruchten01] 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:

- The absence of fundamental theories, or at least practically applicable theories, makes it difficult to reason about software without building it.
- The ease of change encourages changes in software, but it is hard to predict their impact.
- The rapid evolution of technology does not allow proper assessment, and makes it difficult to maintain and evolve legacy systems.
- The very low manufacturing costs combined with ease of change have led the software industry into a fairly complex mess.

An early definition of the term "software engineering" was proposed in 1968 at the first NATO conference on software engineering [Naur68]. Early experience in building software systems, even before the conference, showed that software development was regarded as problematic. This was due to the fact that projects were running late, the cost of software was much higher than predicted, and the software was unreliable and difficult to maintain. The software industry has always been looking for new and effective strategies to develop quicker, cheaper and better software, delivered on time and as budgeted. However, our ability to develop and maintain such software is still inadequate. This is known as the "software crisis", a term that was first used in the end of the 1970s to describe "*ever increasing burden and frustration that software development and maintenance have placed on otherwise happy and productive organizations*" [Griss93, p.549]. Several techniques and methods have been proposed, such as object-oriented technology, Computer-Aided Software Engineering (CASE) tools, formal methods, automatic testing, Model Driven Architecture (MDA) and Aspect-Oriented Programming (AOP) [Mohagheghi04b]. After all these years with software development, the software industry has realized that there is no single "ideal" approach to solve this problem. This is due to the unexpected software evolution (accumulated changes) during development and maintenance of software, both regarding products and related processes. That is, user expectations, technologies, personnel, companies etc. are in a state of constant change. Additionally, Sommerville [Sommerville04] lists three key challenges in software engineering:

- *Heterogeneity*: integrating new software with older legacy systems written in different programming languages, and providing techniques for building dependable software that is flexible enough to handle this heterogeneity.
- *Delivery*: reducing time-to-market for large and complex systems without compromising system quality.
- *Trust*: developing techniques that demonstrate that software can be trusted by its users.

In order to address these challenges, as well as tackle the dynamic nature of software development, we need new technologies and innovative ways of combining and using existing technologies. Hence, one of the new proposed strategies is to assemble, acquire and integrate reusable components, i.e., Component-Based Software Engineering (CBSE).

Since software reuse has been discussed for decades, this raises two questions:

- What is new in CBSE?
- Why will it work now?

The answer to the first question is the focus on software architecture as a guideline to put pieces together [Bass00] [Bachman00], viewing components as independent units (e.g. easy to replace or modify a component without affecting other components) of development and deployment, and assembling applications from sets of those components. The answer to the second question is that the underlying technologies have matured, the business and organizational context where these applications are developed, deployed and maintained has changed, and there is now extensive use of the Internet for sharing and gaining knowledge and experience [Brown98] [Ayala07].

CBSE has helped organizations to develop flexible and maintainable software systems, by offering high-level abstractions, separation of concerns, and encapsulating and hiding complexity in components. This thesis considers how CBSE works in industrial software systems.

2.2 Software Quality

IEEE 610.12-1990 [IEEE90] defines software quality as:

"the degree to which a system, component, or process meets customer or user needs or expectations..."

The first elaborate studies on software quality appeared in the late 1970s [McCall77] [Boehm78]. These studies investigated a number of aspects in software systems, which somehow were perceived to be related to software quality. Since then, different taxonomies on software quality attributes have been presented. The various factors that relate to software quality are hard to measure. Various people will have different perspectives on the quality of a software system, which makes quality even harder to measure quantitatively [Vliet01]. McCall's taxonomy from 1977 is among the first [McCall77], and presents two levels of quality attributes: (1) those that can be measured directly and (2) those that are external attributes and can only be measured indirectly. Even though we can measure reliability, for example by the number of defects encountered so far, the main challenge relies on whether we can claim that lower defect density improves reliability or not.

ISO 9126 [Vliet01] defines a set of quality characteristics and sub-characteristics. Both ISO 9126 and IEEE 610.12-1990 standards are used in this thesis, since they constitute different purposes. The former is a standard of glossary for software engineering terminology, while the latter is a standard for the evaluation of software quality. Table 3 gives a brief overview of all the quality characteristics and sub-characteristics, but only two of these quality characteristics, reliability and maintainability, have been elaborated. Only these two have been investigated indirectly in our studies by defect density and change density respectively. However, we cannot conclude anything about what impact defect density has on reliability, since measuring actual reliability (e.g. mean-time-to-failure) has not been the main focus of this thesis (see Section 1.3). Defect density and change density are related to the sub-characteristics fault tolerance and stability (marked in italics in Table 3). The remaining quality characteristics and sub-characteristics are not studied in this thesis.

Table 3 presents the ISO 9126 quality characteristics and sub-characteristics, but only the description and indirect measure for reliability and maintainability are commented on here.

Table 3. Quality characteristics [Vliet01]

Characteristic	Sub-characteristics	Description	Indirect measure
Reliability	Maturity ***Fault tolerance*** Recoverability	**Fault tolerance**: The capability of the software to maintain a specified level of performance in case of software faults or of infringement of its specified interface.	Defect density
Maintainability	Analysability Changeability ***Stability*** Testability	**Stability**: The capability of the software to minimize unexpected effects from modifications of the software.	Change density
Functionality	Suitability Accuracy Interoperability Security		
Usability	Understandability Learnability Operability Attractiveness		
Efficiency	Time behaviour Resource utilization		
Portability	Adaptability Installability Co-existence Replaceability		

A quality attribute that is not defined in the ISO 9126 standard is *dependability (degree of trust),* a term proposed by Laprie and later by [Avizienis04] to cover the aggregated quality attributes such as availability, reliability, safety, integrity and maintainability. Furthermore, [Sommerville98, p.3] claims that a *"repeatable process that is oriented toward defect avoidance, is likely to develop a dependable system"*.

A *fault* is a dormant static inconsistency (i.e. incorrect with respect to the stated functional requirements) in a software system. *Error* is used to denote the dynamic execution of a passive fault, and may lead to incorrect internal behaviour and system state [IEEE90]. Error is also used for any fault or failure resulting from human activity [Endres04]. The dynamic execution of a fault may however lead to an observable, operational *failure* [IEEE90]. Failures observed by test groups or users are reported back to the developers through *failure reports.* Sometimes, *defect* is used in place of fault, without distinguishing the human/machine origin or whether it is active or passive. This term will be used throughout this thesis to denote failure (after fault executions) or "similar" misbehaviour that technically is not a failure (such as an operational misunderstanding). Other changes to the software can be thought of as *non-defect changes* (as opposed to localizing and fixing defects to maintain status quo). The term *non-defect changes* will also be used throughout this thesis (see Figure 1 for the software change definition in this thesis, and Figure 8 for the software change process in StatoilHydro ASA).

The quality foci in this thesis have been *defect density*, defined as the number of trouble reports divided by lines of non-commented source code (NSLOC), and *stability* measured as *change density*, defined in **P3** and **P6** as:

- Number of change requests (perfective, preventive and adaptive changes) divided by total lines of non-commented source code (NSLOC), collected from Rational ClearQuest. Even though we called it change density in **P3** it is actually *change request density*, and we will use the term change request (CR) density when talking about paper **P3** throughout this thesis.
- Number of changes (corrective, perfective, preventive and adaptive) divided by total non-commented source code (NSLOC), collected from Rational ClearCase. We called it *change density* in **P6**, and we will continue using this term when talking about change density related to **P6**.

See Figure 4 for all the files that make up our code base in this thesis. All of the three Java files A, B and C (which have been changed), and the Java file D (which has not been changed) belong to the same software system, called "JEF" in Figure 4. The total NSLOC for JEF (including Java files A to D) is calculated based on the last changes made to the whole software system[2].

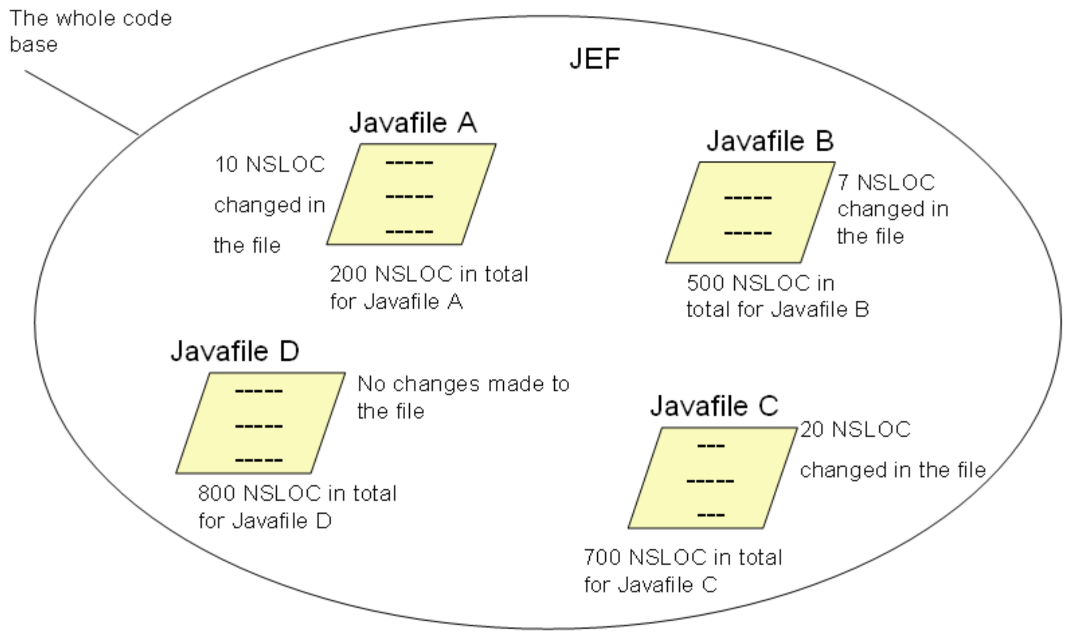
The whole code base
JEF
Javafile A
10 NSLOC changed in the file
200 NSLOC in total for Javafile A
Javafile B
7 NSLOC changed in the file
500 NSLOC in total for Javafile B
Javafile D
No changes made to the file
800 NSLOC in total for Javafile D
Javafile C
20 NSLOC changed in the file
700 NSLOC in total for Javafile C

Figure 4. The whole code base

We have chosen to measure defect density, CR density and change density as these attributes are part of the stated **quality focus** for the reuse program in StatoilHydro ASA. Regarding the discussion about how defect density, CR density and change

[2] The way we have calculated CR density and change density is not ideal. It would have been more precise to "divide" it by the NSLOC when the respective CR and source code change happened, i.e. not by the final NSLOC. However, this was not possible, since we only had the final NSLOC. The resulting consequence is that all these densities have too low numerical values, assuming code growth.

density impact quality (for instance reliability, usability, performance, etc.) we focus on the following factors:

The profile of defects under testing: To discuss testing we can ask ourselves "*if we want to reduce the number of induced failures in software systems, how much testing would be expected, and vice versa*?" One of the important characteristics of software is its reliability, i.e. "*the probability that software will not cause a failure of a system for a specified time under specified conditions"* [IEEE90]. The software system may have defects that do not lead to incorrect behaviour, since they may never be executed, given the actual time interval and usage profile. Many such defects will remain in the software system unknown to the developers and users. That is, a software system with few *discovered* defects is not necessarily the same as a system with few defects [Bertolino07]. High *defect density* before delivery may be an indicator of extensive testing rather than poor quality of the software system [Fenton00a], since most relevant defects might have been eliminated. Hence, it may be impossible to discover the "last" defect in testing, but by using the operational profile to drive testing we might be able to eliminate those defects which would appear more frequently [Bertolino07]. Thus, it is of equal importance to investigate the effectiveness of testing (unit testing, system testing, etc.). For future challenges in reliability testing we refer to Lyu's roadmap [Lyu96].

Value-added defect handling: One of the first software reliability models was introduced by Jelinski-Moranda [Jelinski72]. The model implies that each defect removal has the same lowering effect on the failure rate of the software. Moreover, the failure rate remains unchanged between two successive failures. Thus, when a failure has occurred and the corresponding defect has been removed without introducing new defects into the software, the failure rate decreases by a fixed amount (i.e., the failure rate decreases exponentially). According to Adams [Adams84], the benefit of removing a given defect depends on how many problems it would otherwise cause. He also claims that, after a software system has matured over time, it is not cost-effective to eliminate the majority of the remaining defects as they are mostly irreproducible or of minor nuisance. Boehm [Boehm06] presents an example of test cost savings (on project level as well as on global scale) by focusing testing on the most valuable tests, in relation to the operative profile. The example illustrates that with a cost-benefit focus on testing, thorough testing of the system requires a constantly increasing effort per defect removal. Also, a "cut-off" point will be reached, where further testing is counter productive. Basili and Selby [Basili87] have compared the effectiveness of three state-of-the-practice defect detection techniques; code-reading, functional testing and structural testing. The study compares three aspects of these techniques: defect detection effectiveness, defect detective cost, and classes of defects detected. The authors discuss the outcome that spending more time on detecting defects had no relationship to the amount of defects detected. Therefore, the amount of testing to be conducted should be discussed by profile of the software system (users, the operative environment, etc.) and the context of the company.

Impact of defect density and change request density on reliability: The study by Li and Smidts [Li03] shows that several factors, such as high defect density and high change request density, are considered to have a negative impact on the reliability of software [Li03]. These factors are, however, based on expert opinions rather than measurements, and none of the experts were from the application domain studied in this thesis. The authors mention some limitations to their study (i.e. they were not able to assess the bias in the experts' inputs, and it would have been more comforting to have a larger set of experts). In spite of its limitations, this paper seems to be the only one that tries to show the connection that high defect density and high change request density have a negative impact on reliability. Even though the results in [Li03] show that defect density and change requests correlate significantly with reliability, this does not indicate whether or not this is causality. Therefore, defect density, as well as CR density and change density, cannot be used as standard measures of quality, but remaining defects after testing may impact reliability.

Operational profile testing: Software reliability models were designed to quantify the likelihood of software failure [Jelinski72] [Ramamoorthy82] [Musa87] [Lyu96]. It is also advocated in the Cleanroom development process, which applies the Cleanroom software-engineering method (i.e. statistical test approaches of profile driven testing) in environments that require extensive code reuse [Poore93]. Further, *making a good reliability estimate depends on testing the product as if it were in the field* [Musa96, p.167]. Hence, usage profile is central to determine a system's reliability [Musa96]. As observed in paper **P6**, the change profile is affected by the user profile. Since two new users applied the system and did a lot of acceptance tests, several non-defect changes (e.g. missing functionalities) were implemented, but how these changes impact reliability we cannot say since we have not measured it.

Discussion: Most of the software reliability models are based on software failure observations made during test or operation, and they assume that the operational profile and remaining defects in the software impact reliability. *In those cases where the companies decide to measure attributes (e.g. complexity, defect density) then software reliability may have to be* **indirectly** [bold added by us] *assessed from available sets of software engineering measures* [Li03, p.811]. However, measuring actual reliability, e.g. mean-time-to-failure, has not been the main focus of this thesis (see Section 1.3), due to the available data set. Based on our discussion here we assume that defect density, CR density and change density may be indirect indicators of reliability, but we do not have any measured data in this thesis or have found any relevant studies to confirm this. Therefore, in this thesis we do not give more attention to the factors that might impact reliability, but it is a part of the future work (see Section 7.4.1).

2.3 Software Reuse

Reuse-based software engineering is a strategy where the key concept is to reuse existing software. Doug McIlroy was the first to introduce the idea of systematic reuse (the planned development and widespread use of software components) at the above-mentioned NATO software engineering conference in 1968 [McIlroy69]. Morisio et al. [Morisio02] define software reuse as:

… 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.

This definition is the one used in the thesis, since this work focuses on the investigation of a reused framework vs. software reusing it.

Several software organizations around the world have reported on successful reuse strategies and programmes, such as IBM, Hewlett Packard, AT&T, Toshiba and many others [Griss93]. These reports show that reuse actually "works" and the achieved benefits are substantial in form of improved quality and productivity, reduced time-to-market, a standardized architecture, and/or reduced cost. Reuse is possible at different levels, encompassing several approaches and situations [Karlsson95]. According to [Mohagheghi04b] the reusable assets or components can be subroutines or classes in library, free-standing COTS (Commercial-Off-The-Shelf) or OSS (Open Source Systems) components, modules in a domain-specific framework (e.g. Smalltalk MVC classes), or entire software architecture and their components, forming a product line or a product family. One of the most effective reuse approaches in the industry is to use a *product line* or *application families*. A product line is a set of applications with a common application-specific architecture [Sommerville04]. The key idea is to define a common architecture and a set of core components/assets that can be reused. Thus, each time a new application is developed, the common core of the application family is reused almost "as-is".
Reusability has been defined as a combination of two characteristics [Karlsson95] [Mili02]:

- *Usefulness (generality)*: to what extent the functionality of a component is needed.
- *Usability (understandability)*: to what extent a component is packed for reuse.

So, when designing a component for reuse, there is a trade-off between generality and understandability. Johnson and Foote [Johnson88] claim that software reuse does not happen by accident. The system designers must plan to reuse old assets and look for new reusable assets. According to the authors, reusable classes are discovered, not designed. Many factors may influence the success or failure of software reuse, and developing for reuse has its price. It is therefore necessary to investigate which factors contribute positively to implement a reuse programme, so that organizations can increase their chances to succeed. Morisio et al. [Morisio02] did a survey using structured interviews, where they analysed 24 EU projects on software reuse in large and small companies in Europe in the years 1994-1997. The projects vary in size, development approach, type, etc., and few of them have defined their own reuse metric. The results revealed that successful software reuse was achieved when the development organization had a potential for reuse because of:

- commonality among applications,
- top management committed to introducing reuse processes,
- modifying non-reuse processes, and
- addressing human factors (e.g. reuse education).

However, size, experience, development approach (object-oriented or not), actual implementation language, reward policy, asset repository and reuse measurement were not found to be decisive factors for successful software. In the end, the above authors concluded that reuse approaches vary and should be adjusted to the context of the company.

Frakes and Fox [Frakes95] conducted a reuse survey in 1991-1992, where they posed 16 commonly asked questions about reuse. A total of 113 people from 28 US organizations and one European organization, with a median size of 25,000 employees, participated in this survey. The results revealed that education influences reuse, that developers actually prefer to reuse instead of build components from scratch, that reuse is more common in telecommunications compared to aerospace, and that having a reuse repository is not critical for software reuse. Additionally, they found that a common defined software process may be advantageous.

According to prior research [Griss95] [Morisio02], systematic reuse does not just happen, but must be planned and introduced through an organization-wide reuse programme. Griss and Wosser [Griss95] claim that non-technical issues (process, management, organization etc.) are more important than technical ones (standards, architecture, framework etc.). They also mention the three most critical elements that reuse needs, namely:

- *Management support*: ongoing involvement and support from managers are important because reuse now involves several projects.
- *Common wisdom*: object-technology and libraries are not essential parts of reuse. It is more important to have an explicit reuse agenda. Hence, domain stability and experience are often more important for successful reuse than general process maturity.
- *Incremental adoption*: as you focus on reuse and learn more about your process, the levels of reuse will increase.

Sommerville [Sommerville04] lists the following challenges within software reuse:

- *Increased maintenance cost*: if the source code of the reusable assets is not available, it may become incompatible with the system change.
- *Lack of tool support.*
- *Not-invented-here syndrome*: not trusting other people's software.
- *Creating and maintaining a component library.*
- *Finding, understanding and adapting reusable components.*

2.3.1 Studies on Software Changes and Software Reuse

The related work presented in this section has been cited verbatim from our own research papers **P5** and **P6** in Appendix A (see the specific references in text where they apply).

Our paper **P5** (see Appendix A) summarizes studies that have compared the defect densities of reused components with non-reused components, as shown in Table 4 (the table is found in paper **P5** in Appendix A, p.144). These studies were a result of a systematic survey done by Mohagheghi et al. [Mohagheghi07]. Results from these studies show that continued reuse with slight modification to components results in significantly lower defect/problem density and significantly less effort expended on development and/or correction.

Table 4. Studies related to defect density and reuse (P5 in Appendix A, p.144)

Quality focus	Quality measures	Conclusion
Reusable vs. non-reusable components [Lim94]	No definition of what a defect is. Defect density is given as defects/1000 non-comment source statements (KNCSS).	Reuse can provide improved quality, increased productivity, shortened time-to-market, and enhanced economics.
Reusable vs. newly developed components [Thomas97]	Error/defect densities (errors/defects per 1000 source statements). However, no definition of error/defect.	Reuse provides an improvement in error density (more than a 90% reduction) compared to new development.
Reusable vs. non-reusable components [Frakes01]	Error density (number of errors per non-commented line of code) from the pre-delivery stage of the system.	More reuse results in lower error density.
Code reuse [Succi01]	-Client complaint density (i.e., the ratio of client complaints to lines of code) -Defect density after the system is delivered to the client	Reuse is correlated significantly and positively with client satisfaction.
Reusable vs. non-reusable components [Mohagheghi04c]	Defect density (number of defects/lines of non-commented code)	-Reused components had lower defect density than those that were not reused. -Reused components had a higher number of defects of the highest severity before delivery, but fewer defects post-delivery.
Reused, modified and newly developed modules [Selby05]	Module fault rate (number of faults in a module per non-commented source lines of code). Since an error correction may affect more than one module, each module affected by an error is counted as having a fault.	-Software modules reused without revision had the fewest faults, fewest faults per non-commented source line of code, and lowest fault correction effort. -Software modules reused with major revisions had the highest fault correction effort and highest fault isolation effort.

Some studies have proposed explanations for the lower defect density of reused components. For example, Lim [Lim94] proposed the following: 1) as work products are used multiple times, the defect fixes for each reuse accumulate, and gradually result in higher quality, and 2) more importantly, reuse provides incentives to prevent and remove defects earlier in the life cycle because the cost of prevention and debugging can be amortized over a greater number of uses. Succi et al. [Succi01] proposed that implementing a systematic reuse policy, such as the adoption of a domain-specific library, improves client satisfaction. Selby [Selby05], Frakes et al. [Frakes01], and Thomas et al. [Thomas97] attributed the lower defect density of reused components to the smaller number, and fewer amounts, of changes performed on them. In addition, Thomas et al. [Thomas97] proposed the following: 1) if there is an expectation that components will be reused, it is more likely that they will be well-specified, particularly with respect to their reuse functionality; 2) the nature of the programming languages, i.e. FORTRAN and Ada in their cases, may affect the benefits of reuse, and 3) the experience with reuse in an organization and the approach taken towards reuse are likely to influence the nature of defects. A close examination of these studies illustrates that (paper **P5** in Appendix A, p.144):

- Most studies compared only the number of defects between reused and non-reused components without going into further detail. The one exception is Thomas et al. [Thomas97], who divided the defects into defect types and compared the number of defects of each type. However, no studies have so far investigated differences in defect densities in reused components with respect to the type of defect (paper **P5** in Appendix A, p.144).
- Many factors may influence the success or failure of software reuse [Morisio02] [Rothenberger03], such as management commitment, the process by which reuse is introduced, and human factors. It is therefore necessary to investigate which factors contribute positively to the lower defect density of reused software, and which contribute negatively. In addition, it is important to understand which factors need to be excluded before analysing the relationship between software reuse and lower defect densities of reused software. Some studies [Lim94] [Thomas97] [Frakes01] [Succi01] [Selby05] have attempted to attribute the lower defect densities of reused vs. non-reused software to the practices of reuse. However, few of them have done convincing cause-effect analyses (paper **P5** in Appendix A, p.145). Most of them simply proposed possible explanations without providing confirmation, as shown in Figure 5 (the figure is from paper **P5** in Appendix A, p.145).

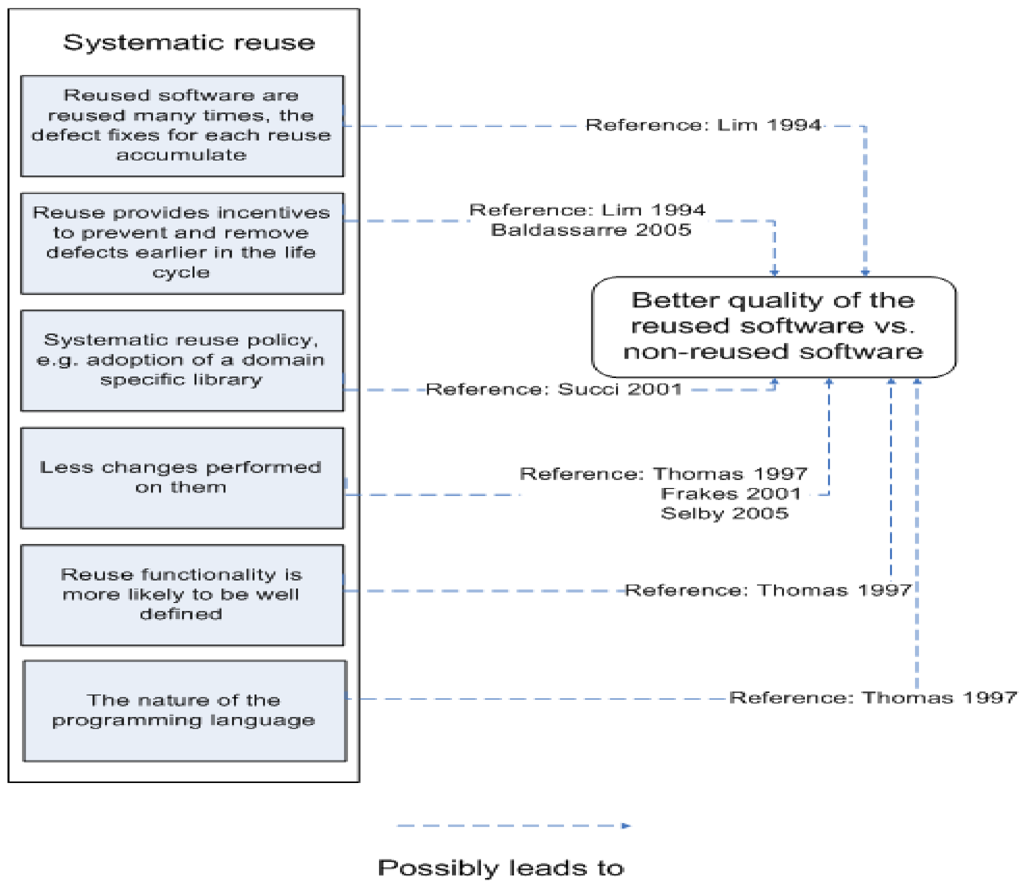
Systematic reuse
Reused software are reused many times, the defect fixes for each reuse accumulate
Reuse provides incentives to prevent and remove defects earlier in the life cycle
Systematic reuse policy, e.g. adoption of a domain specific library
Less changes performed on them
Reuse functionality is more likely to be well defined
The nature of the programming language
Reference: Lim 1994
Reference: Lim 1994 Baldassarre 2005
Better quality of the reused software vs. non-reused software
Reference: Succi 2001
Reference: Thomas 1997 Frakes 2001 Selby 2005
Reference: Thomas 1997
Reference: Thomas 1997
Possibly leads to

Figure 5. Current research proposals regarding the overall cause-effect relationship between software reuse and the lower defect/error density of reused software (P5 in Appendix A, p.145)

Another study by Ostrand et al. [Ostrand05] has studied defect distributions among two industrial systems. Even though the authors do not mention explicitly reuse impact as their central focus, they do study number of defects across subsequent releases. A negative binomial regression model using information from previous releases has been developed and used to predict the numbers of defects for two large industrial systems [Ostrand05]. The predictions were quite accurate for each release of the two systems, correctly selecting files that contained between 71% and 92% of the defects that were actually detected, with the overall average being 83%. The information about the systems also shows that the defect density (defects per KLOC) tends to decrease as the system matures.

Our paper **P6** (see Appendix A) has summarized studies [Frakes01] [Algestam02] [Mohagheghi04c] [Selby05] that have examined the possible influence of software reuse on the changes of a system, as shown in Table 5 (the table is found in paper **P6** in Appendix A, p. 170).

Table 5. Studies comparing changes of reused components vs. those in non-reused ones (P6 in Appendix A, p.170)

Quality focus	**Quality measures**	**Conclusion**
Change density	Number of change requests per source code line.	Reused components are more stable in terms of volume of code modified between releases [Mohagheghi04c].
	Percentage of code-line changes (enhancement or repair).	The modules reused with major revision (>=25% revision) had the most code changes per SLOC [Selby05].
Number of changes	The number of changes (enhancement or repair) to a module.	More reuse results in fewer changes [Frakes01].
Amount of modified code	Size of modified or new/deleted code/total size of code per component between releases.	Non-reused components are modified more than reused ones [Mohagheghi04c].
Number of change scenarios	Number of changes to which a software system is exposed (e.g. adding communication protocols, porting to new platforms, issues related to the database manager)	Reusing components and a framework resulted in increased maintainability in terms of cost of implementing change scenarios [Algestam02].

Although most studies [Frakes01] [Algestam02] [Mohagheghi04c] in Table 4 and 5 conclude that *software reuse is significantly correlated to fewer changes or lower defect density*, one study observed that a reused module that undergoes major revision has the most changes per source line [Selby05]. A close investigation of the studies in Table 4 and 5 further illustrates that (paper **P6** in Appendix A, p.170):

- None of the studies performed detailed analyses (as is the case with studies listed in Table 6 in Section 2.5.1, from **P6** in Appendix A). "Detailed analysis" here refers to dividing the changes into different types and comparing the distribution of the changes according to type. Several factors (e.g. complexity, functionality, development practice, age, and size) may determine the profile of software maintenance [Kemerer97]. Thus, *comparing only the number or density of the defects is not sufficient to warrant the conclusion that software reuse is significantly correlated to fewer changes* (paper **P6** in Appendix A, p.170).

Thus, further study is needed to investigate the relation between software reuse and software changes of different types for reused and non-reused software, which is done in this thesis.

2.4 Component-Based Software Engineering

Over the past few decades, several attempts have been made to improve software development practice. Some of the approaches have been improved design techniques, developing more expressive notations for capturing an intended functionality of a system, as well as encouraging reuse of pre-developed component pieces rather than developing from scratch [Brown98]. Already in 1972, Davis Parnas wrote about the advantages of decomposing a system into modules [Parnas72]:

- shorter time-to-market (development time) because modules can be developed in parallel by separate groups,
- increased product flexibility,
- ease of change, and
- increased comprehensibility as modules can be studied separately.

A new paradigm for software development that emerged in the 1990s was CBSE, which represents development *with reuse* in contrast to development *for reuse* [Karlsson95]. The latter refers to systematic generalization of software components for later reuse, while the former deals with how existing components can be reused in existing or new software systems. Regarding the reuse of "in-house" components, these two development processes are tightly related.

The creation of CBSE emerged from designers' frustrations that object-oriented principles did not lead to extensive amount of reuse, as originally suggested [Sommerville04]. The reason was that individual object classes: (1) were too detailed and specific, (2) were often bound to an application at compile-time, (3) required detailed knowledge of the classes, which implied access to the source code, and (4) were dependent on several other super classes. All this made it difficult to reuse individual object classes. CBSE, however, assumes more high-level units for assembly, and makes systematic reuse possible by demanding that components should adhere to a component model. CBSE is the process of defining, implementing, composing and integrating loosely coupled independent components into systems.

Both CBSE and *Component-Based Development* (CBD) are approaches to the old problem of handling system complexity by decomposition, and these two approaches are often used indistinguishably. Although, much effort has been devoted to define and describe the terms and concepts involved, there is literature [Bass00] that distinguishes between these two concepts. According to Bass et al. [Bass00], CBD involves the technical steps for designing and implementing software components, assembling systems from pre-built software components, and deploying assembled systems into their target environment. CBSE then involves the practices necessary to perform CBD in a repeatable way to build systems that have predictable properties [Bass00].

CBSE has become an important software development approach, and has gained much attention in the software industry, as a way to handle software complexity and evolution in a cost-effective and quality-ensuring way. The following are some of the benefits of using CBSE [Bachmann00] [Bass00]:

- *Independent extension*: components are units of extension, and a component model describes exactly how extensions are made.
- *Component markets*: deployment of components into a common environment.
- Components improve *programmer productivity*, which also reduces *time-to-market*: the availability of components reduces the time it takes to design and develop systems. Even if component families are not available in an application domain, the uniform component abstractions will reduce development and maintenance costs overall.
- *Separation of skills*: complexity is packaged into the component framework, and provides distinct services for developers, assemblers, deployers and administrators.
- *Components provide a base for reuse*: components have direct usability - they can be applied directly to build a system in contrast to design patterns or other more abstract forms of packaged reuse, which require adaptation before providing usability.

In the literature, components are defined and classified in several ways. Definitions vary according to the following factors [Mohagheghi04b]:

- life cycle phase for component identification, e.g. high level abstractions vs. implementation units,
- origin; in-house, bought (COTS) or free software (OSS), or
- roles a component can play in a system, such as process components and data components.

One of these component definitions is presented below. Szyperski [Szyperski02] defines a *component* as:

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.

The terms *component model* and *component framework* are also used when talking about CBSE, but are often intermixed. Bachmann et al. [Bachmann00] define a component model as a "*set of component types, their interfaces, and a specification of the allowable patterns of interaction among component types*". They also define a component framework as a "*framework that provides a variety of runtime services to support and enforce the component model and component interaction*". Component frameworks are like special-purpose operating systems, but operating at much higher levels of abstraction. Hence, developing a component framework is demanding. Some examples of commercial component frameworks (also called component technologies) are EJB, .NET, COM+, and CORBA.

CBSE is therefore about developing components based on a component model and composing components into application systems. Important aspects are **reuse, autonomy** of components, and **composition** [Mohagheghi04b]. According to Sommerville [Sommerville04], there is a trade-off between reusability and usability of a component. To make a component reusable involves providing a comprehensive set of generic interfaces and operations that can cater to all the ways in which the component could be used [Sommerville04]. So, adding generality to a component increases its reusability, but decreases the usability. The more operations a component has, the more “complex” it gets, and this makes it more difficult to understand and use. It is therefore important to find a compromise between generality and understandability, when designing a component for reuse.

Although CBSE is rapidly being adopted as a mainstream approach to software development, some challenges still remain. Bass et al. [Bass00] mention the following challenges or inhibitors in CBSE, in decreasing order of importance: lack of available components, lack of stable standards for component technology, lack of certified components, and lack of an engineering method to consistently produce quality systems from components. Crnkovic [Crnkovic02] lists the main concerns of CBSE related to components as: component specification, its implementation, and its deployment. He also talks about the three challenges CBSE faces when dealing with extra-functional properties (referring to quality attributes and non-functional requirements): (1) inaccurate definition of these properties, (2) difficulty of relating the overall system properties to component properties, and finally (3) current component-based technology has no satisfactory support for specifying these properties. Another researcher [Voas01] mentions the difficulty of combining quality attributes (e.g. non-functional requirements) related to component use since users do not know all the quality attributes a component may need and how the system will tolerate these attributes.

Sommerville [Sommerville04] claims that the long-term vision of CBSE is that there will be specialized component suppliers whose business is based on the development and sale of reusable components. However, it is unlikely that this vision will be realized before these aforementioned challenges have been satisfactorily solved.

2.5 Software Evolution and Software Maintenance

Observations and rules relevant to *software system evolution* planning and management were first identified during studies of evolution of OS/360-70 and other systems between 1968 and 1985. The majority of work in this area was conducted by Lehman and Belady, in the 1970s and 1980s [Lehman85]. It was, however, Meir M. Lehman who was the first to become aware of this phenomenon and who did the first systematic studies of software evolution, while he was working for IBM in the late 1960s. He realized that the real problem was not in the architecture of a specific system, but rather in the methods and methodology of the architecture.

In the 1990s Lehman and other researchers investigated the significance of feedback in evolution processes [Lehman96] [Lehman98a] [Lehman01a]. The results from these studies were the *Lehman Laws* for system change. Lehman and Belady have extensively studied the growth and evolution of a number of large software systems. Based on those quantitative studies they first proposed five laws: (I) Continuing Change, (II) Increasing Complexity, (III) The Fundamental Law of Program Evolution, (IV) Conservation of Organizational Stability, and (V) Conservation of Familiarity [Belady76]. These laws were further developed and revised through the FEAST/1 [Lehman98b] and FEAST/2 [Lehman01b] research projects.

Lehman Laws serve as a guide to the evolutionary software development process and the construction of software tools. Lehman and Ramil [Lehman01c] distinguish between two types of systems, E-type and S-type systems. The former is defined as: "*software used for problem-solving or application-addressing in a real-world domain*". This type of software is accepted based on aspects such as user satisfaction, functionality, and performance, and cannot be proven correct [Lehman01c]. The S-type system focuses on the problems that are formally defined and specified. According to Lehman [Lehman05], a program derived from such specifications will be required to be correct, with respect to a fixed and consistent specification. However, the laws apply primarily to the E-type systems, and Lehman discovered that all such systems undergo evolution that will affect all aspects of the system. In essence, this means that evolution is continuous and pervasive, and gradually changes the scope and architecture of a software system (often called a legacy system), until such a system is no longer economically maintainable in terms of available resources (effort, time etc.) [Lehman02].

The first formulated laws of software evolution were not widely accepted as relevant to software engineering practice. This was due to the absence of precise definitions of the laws, lack of significant support for some of the laws when applying statistical tests, and finally about using the term "law" to characterize human-social phenomena activities [Lehman01c]. However, over the years they have been recognized, as they provide useful inputs to understand the software processes. The increased awareness of software evolution is, according to Lehman and Ramil [Lehman01c], due to several factors, such as the pervasiveness of computers, their growing deployment in industry, commerce, government etc., and the increased use of the Internet.

The term software evolution is used in different ways by various researchers, and there is no overall agreement on a definition. Some researchers use the term to encompass both the initial development of the system and its subsequent maintenance [Sommerville01], others use it exclusively to refer to the events after initial implementation [Kemerer99]. Bennett and Rajlich [Bennett00] see evolution as a separate stage in the lifecycle of the software. One common definition of software evolution, proposed by Belady and Lehman [Belady76] is:

Software Evolution: "*….the dynamic behaviour of programming systems as they are maintained and enhanced over their life times….*"

Today's software systems in organizations are becoming more long-lived and hence evolution is becoming of particular importance. However, studying software evolution seems to be challenging due to the longitudinal nature of the phenomenon, difficulties in collecting empirical data, and the lack of theory and models [Kemerer99]. *How can empirical software researchers address the evolution phenomenon?* Several authors such as [Kemerer99] [Godfrey00] [Lehman01c] believe that one way of handling this phenomenon is to provide researchers with more understanding of the **what** and **why** of evolution. The focus is on the properties of the phenomenon, its causes and identification of its drivers, and its maintenance activity. An alternative view adopted by [IWPSE01], for example, is primarily concerned with the **how** of evolution, referring to the technology used to allow a more systematic, controlled and efficient software change in system characteristics (e.g. functionality) and growth. Bennett and Rajlich [Bennett00] claim that software evolution needs to be evaluated as a business issue as well as a technology issue, and therefore is fundamentally interdisciplinary. The "grand challenge" within software engineering seems to be the ability to change and evolve software with sufficient ease and quality. However, many systems, especially older legacy systems, are difficult to understand and change. A way of changing such a legacy system is to improve their structure and understandability through *re-engineering*, which is one of the processes in software evolution [Sommerville04]. This means that the functionality of the system, and most of the time the architecture of the system, remains unchanged.

In the literature we can see that software evolution is strongly related to *software maintenance,* which in IEEE Standard 1219 [IEEE93] is defined as:

Software Maintenance: "*….the process of modifying a component after delivery to correct faults, to improve performance or other attributes, or to adapt to a changed environment*".

While maintenance refers to activities that take place at any time after the new development project is implemented, software evolution is defined as examining the dynamic behaviour of systems, and how they change over time. However, the definition of software evolution by Belady and Lehman [Belady76] presented earlier, shows that software evolution can be seen as a much broader term than software maintenance.

During their lifetime, software systems usually need to be altered and the original requirements may change to reflect changing business, user and customer needs [Postema01]. Therefore, software evolution is incorporated into *corrective, adaptive, perfective and preventive* software maintenance [Bennett00] [Postema01]. The definitions and classifications for the different types of maintenance activities vary amongst practitioners in the field. However, we present those defined by Kitchenham and Sommerville [Kitchenham99] [Sommerville01], which are also used throughout this thesis. These are:

- *Adaptive* changes are those related to adapting to new platforms, environments or other applications.
- *Corrective* changes are those related to fixing bugs.
- *Perfective* changes are those that encompass new or changed requirements as well as optimizations.
- *Preventive* changes are those having to do with restructuring and reengineering.

The definitions of maintenance activities used throughout this thesis are also similar with the ones in [Bennet80] [Fenton96]. In fact, the adaptive and perfective parts of software maintenance can be thought of as part of software evolution [SEVO04]. That means, it can encompass environmental adaptations as well as both aspects of modified and added scope. Platform changes, on the other hand, are sometimes referred to as porting, rather than software evolution [Frakes95]. Still, there is the opposing view that software maintenance starts prior to delivery of the software.

Sommerville [Sommerville01] defines maintenance as the single most expensive activity in software engineering, requiring 65% to 75% of total effort. Therefore the costs of these maintenance operations are additionally higher than that of developing the original software. There are several contributing factors for this, but one of the most important seems to be that maintenance staff is inexperienced or unfamiliar with the domain [Postema01]. A pioneer study conducted by Lientz et al. [Lientz78] showed that around 75% of the maintenance effort was on adaptive and perfective activities, and error correction consumed about 21%. This study shows that the incorporation of *new user requirements* is the core problem for software evolution and maintenance. Therefore, progress in software architecture is crucial, so that practitioners may extend and adapt functional and non-functional user requirements without destroying the integrity of the architecture. Krogstie et al. [Krogstie06] conducted a survey to investigate the development and maintenance of business software in Norway. The same survey was performed in 1993 and 1998. The results show that the overall time used for maintenance is around 40%.

Two basic factors are given by Bennett and Rajlich [Bennett00] about why software maintenance is important:

- it consumes a large part of the overall lifecycle costs, and
- the inability to change software quickly and reliably means that business opportunities are lost.

According to Bennett and Rajlich [Bennett00], these factors are *enduring problems*, hence the profile of maintenance research is likely to increase over the next ten years.

2.5.1 Studies on the Distributions of Different Types of Software Changes

The related work presented in this section has been cited verbatim from our own research paper **P6** in Appendix A (see the specific references in text where it applies).

Our paper **P6** (see Appendix A) summarizes studies that examine the *static* aspect of software changes, i.e. the distribution of different kinds of change, or the distribution of effort spent on performing different kinds of change. Table 6 (the table is found in paper **P6** in Appendix A, p.166) summarizes the nature and conclusions of these studies.

Table 6. Studies related to distributions of different changes (P6 in Appendix A, p.166)

Study description	Distribution and definition of different types of change	Other observations of the study
A questionnaire-based survey that collected data from 69 systems, which were developed using different programming languages, e.g. Cobol, Assembler, Fortran [Lientz78].	- 60% **perfective** (enhancements and speed performance) - 18% **adaptive** (changes to data inputs and files) - 17% **corrective** (emergency fixes and debugging) - 4% **other** (no description given)	User demands for enhancements and extensions constitute the most important problem area with respect to management.
A case study that investigated change requests collected for two years in a Canadian financial institute [Abran91]. Used the same definitions as [Lientz78] for corrective, adaptive, and perfective changes. Analysed 2152 change requests.	- 60% **adaptive** - 21% **corrective** - 3% **perfective** - 15% **user support** (handle user requests of application rules and behaviour, requests for work estimates, requests for preliminary analysis)	Maintenance team in 1989 spent 64% of their time doing maintenance work (e.g. optimization and adding new functionality) other than correcting defects and errors.
A survey conducted in the MIS (Management Information System) department in nine different application domains in Hong Kong. 1000 questionnaires were sent out and about 50 responses were received [Yip94].	- **perfective** (40% enhancements, 7% tuning, and 6% reengineering) - 16% **corrective** (correct faults) - 10% **adaptive** (adaptation to new environment) - **other** (13% answering questions and 9% documentation)	In Hong Kong, 66% of the total software life cycle effort was spent on software maintenance. The most cited maintenance problems were staff turnover, poor documentation, and changing user requirements.
A structured interview with managers and maintainers in a computer department of a large Norwegian telecom organisation in 1990-1991 (study1)	**Results of interviews with managers:** - 44% **perfective** (changes in user requirements) - 29% **adaptive** (make software usable in a changed environment)	If the amount of corrective work is calculated on the basis of interviews solely with managers, it will be twice as much as the actual work reported in

Study description	Distribution and definition of different types of change	Other observations of the study
and 1992-1993 (study2) [Jørgensen95]. Systems were developed using either Cobol or Fourth Generation languages.	- 19% **corrective** (correct faults) - 8% **preventive** (preventing problems before they occur) **Results of interviews with maintainers:** - 45% **perfective** - 40% **adaptive** - 9% **corrective** - 6% **preventive**	logs (i.e. the amount of corrective work may be exaggerated in interviews).
Studied 10 projects conducted in the Flight Dynamic Division (FDD) in NASA's Goddard Space Flight Center. The FDD maintains over 100 software systems totalling about 4.5 million lines of code. 85% of the systems are written in FORTRAN, 10% in Ada, and 5% in other languages [Basili96a].	- 61% **perfective** (improve system attributes and add new functionality) - 20% **other** (e.g. management, meeting) - 14% **corrective** (correct faults) - 5% **adaptive** (adapt system to new environment)	Error corrections are small isolated changes, while enhancements are larger changes to the functionality of the system. More effort is spent on isolation activities in correcting code than when enhancing it.
A case study investigated the change of maintenance requests during the lifecycle of a large software application (written in SQL) [Burch97]. Analysed 654 change and maintenance requests.	- 49% **repair** (fixing bugs) - 26% **enhancement** (add or modify functionalities) - 25% **user support** (consulting and answering user requests)	User support reaches its peak in the 4th month (first stage). Repair reaches its peak in the 13th and 14th months (second stage), while enhancement is the dominant factor in the third stage (25th month).
A survey carried out in financial organizations in Portugal. Data was collected from 20 project managers [Sousa98].	- 49% **adaptive** (changes in platform) - 36% **corrective** (error modifications) - 14% **perfective** (expand	3% of the respondents considered the software maintenance process to be very efficient, while 70% considered the efficiency to be very

Study description	**Distribution and definition of different types of change**	**Other observations of the study**
	system requirements and optimization) - 2% **preventive** (future maintenance action)	low.
An Ada system of the NASA Goddard Space Flight Center [Evanco99]. Analysed 453 non-defect changes.	- 31% **planned enhancements** (anticipated at the start of development) - 30% **other** (code debugging, enhancements and maintainability) - 29% **requirements modifications** (implementation of requirement changes) - 10% **optimization** (optimize software performance)	Changes related to optimizations require the most effort to isolate, while planned enhancements require the most effort to implement.
A subsystem that contains 2 million lines of source code [Mockus00]. Analysed 33171 modification requests.	- 46% **corrective** (fixing faults) - 45% **adaptive** (adding new features) - 5% **inspection** (code checking to figure errors) - 4% **perfective** (code restructuring)	Corrective changes tend to be the most difficult, while adaptive changes are difficult only if they are large. Inspection changes are perceived as the easiest.
A case study on re-engineering a people-tracking subsystem of an automated surveillance system, which was written in C++ and had 41 KLOC [Satpathy02]. Analysed the distribution of maintenance effort during the whole maintenance phase.	- 38% **perfective** (optimization, restructuring and adding new functionalities) - 31% **adaptive** (adapting to changed environments) - 23% **preventive** (preventing malfunctions and improving maintainability) - 8% **corrective** (correcting problems)	The effort required to adapt the system was high, because the software needed to be ported to a different platform.
Examined three software products: – A real-time product	The analysis and collection of data were performed at two levels, using the same definition as [Lientz78]: (1) **change log**	All three maintenance categories were statistically very highly significantly different

Study description	Distribution and definition of different types of change	Other observations of the study
written in a combination of assembly language and C. Data of 138 modified versions were collected. – The Linux kernel. Data from 60 modified versions were collected. – GCC (GNU Compiler Collection). Data from 15 versions were collected. [Schach03].	**level**, i.e. each entry in the change log was regarded as one unit of change. (2) **module level**, i.e. all the changes made to a module were regarded as a single unit of maintenance. **Change log level:** - 57% **corrective** - 39% **perfective** - 2.4% **other** - 2.2% **adaptive** **Code module level:** - 53% **corrective** - 36% **perfective** - 4% **adaptive** - 0% **other**	from the results of [Litentz78]. Corrective maintenance was more than three times the level of the results of [Litentz78].
Four releases of a telecommunication system written in Erlang, C, Java, and Perl. [Mohagheghi04a]. Analysed 187 change requests.	- 61% **perfective** (new or changed requirements as well as optimization) - 19% **adaptive** (adapting to new platforms or environments) - 16% **preventive** (restructuring and reengineering) - 4% **other** (saving money/effort) Corrective changes are reported elsewhere.	There is no significant difference between reused and non-reused components in the number of change requests per KSLOC.
Web-based Java application, consisting of 239 classes and 127 JSP files [Lee05]. Based on Swanson's definition [Swanson76] and Kitchenham's ontology [Kitchenham99]. Analysed 93 fault reports.	**Based on Swanson's definitions:** - 62% **perfective** - 32% **corrective** - 6% **adaptive** **Based on Kitchenham's ontology:** - 68% **enhanced maintenance** - 32% **corrective**	Maintenance effort of Java application is similar to the distribution in previous non object-oriented and non web-based applications.

A close investigation of studies in Table 6 reveals that:

- *Different studies classify changes differently*, noticed by [Chapin01].
    - Four studies classified changes into four categories: adaptive, corrective, perfective, and preventive [Jørgensen95] [Sousa98] [Satpathy02] [Mohagheghi04b] [Lee05] (paper **P6** in Appendix A, p.168).
    - Several studies did not include preventive changes and classified the changes into adaptive, corrective, and perfective, with a fourth category of user support in [Abran91], inspection in [Mockus00], and “other” in [Lientz78] [Yip94] [Basili96a] [Schach03] (paper **P6** in Appendix A, p.168).
    - One study classified changes into planned enhancement, requirement modifications, optimization, and “other” [Evanco99] (paper **P6** in Appendix A, p.169).
    - One study classified changes into user support, repair, and enhancement [Burch97] (paper **P6** in Appendix A, p.169).
- *Definitions of different types of change are slightly different.* For example, perfective change is defined as user enhancements, improved documentation, and recoding for computational efficiency in [Lientz78], and as restructuring the code to accommodate future changes in [Mockus00]. Perfective change is also defined as encompassing new or changed requirements (expanded system requirements) as well as optimization in [Sousa98] [Mohagheghi04b], and is defined as enhancements, tuning, and reengineering in [Yip94] (paper **P6** in Appendix A, p.169).
- *The distributions of different types of change are not the same for different systems.* 62% of studies, including [Lientz78] [Yip94] [Basili96a], found that perfective changes (the median value of perfective changes of those studies presented in Table 6 is 57%) were the most frequent. However, perfective changes in the system in [Mockus00] were the least frequent. 23% of the studies, reported by [Burch97] [Mockus00], found that corrective changes were the most frequent. 15% of the studies, including [Abran91] [Sousa98], found that adaptive changes were the most frequent (paper **P6** in Appendix A, p.169).

2.6 Summary and the Challenges of this Thesis

Software quality, software reuse, CBSE, and software evolution and maintenance were presented in the previous sections. This section presents those challenges that are relevant for this thesis in the context of reused and non-reused industrial software systems. Some important Research Challenges (RC) can be defined as:

RC1. Indicators of software quality: In the literature, defect density, CR density and change density have been used as a measure for software quality [Mohagheghi04c], but these cannot be used as standard measures. However, we would assume that lower defect density, CR density and change density over successive releases would gradually indicate more stable software. This research challenge is investigated by research question **RQ1** in this thesis.

RC2. Software reuse and its relation to software changes: There are empirical studies that have proposed explanations as to why we observe fewer changes or lower defect density in reused components. However, few of them have done convincing cause-effect analyses. Thus, most of these studies have just proposed possible explanations without further confirmation. This work aims to contribute towards explaining the relation between software reuse and the defect densities for different software change types of the reused software. This research challenge is formulated into research question **RQ1** in this thesis.

RC3. Potential advantages and/or disadvantages of software reuse: "*Software reuse is the systematic practice of developing software from a stock of building blocks*" [Morisio02]. The company will build some components developed for reuse in the beginning. In the case of successful reuse, more and more code will be encapsulated into these components. The components can either be an in-house built component, a bought component (COTS), or free software (OSS). Different types of components have different advantages and challenges. Moreover, the challenges facing reuse and CBSE are also organizational, managerial, and technical (e.g. architectural). Focusing only on the technological issues usually does not bring the whole benefit of reuse and CBSE. This research challenge is studied in research question **RQ3** in this thesis.

RC4. Software evolution and maintenance: There is previous work done on both software evolution and maintenance (see Table 6 in Section 2.5.1), but there is a lack in the literature of empirical studies on evolution and maintenance, comparing the change profile for a reused framework and software reusing it. Software organizations need to understand how their software systems evolve, and have the appropriate processes and resources to manage them, such as requirements handling and SCM. This work aims to empirically study the change profile for a reused framework and software reusing it, and to determine the possible similarities and differences between their change profiles. This research challenge is formulated into research question **RQ2** in this thesis.

3 Research Methods and Metrics

This chapter provides a brief introduction to research approaches and strategies (Section 3.1). It also gives a brief and general description of survey approach and case study approach, the two research methods used in this thesis (Section 3.1). Section 3.2 presents the goal and criteria for defining metrics and types of metrics, as well as the validity threats for all types of studies, and in particular how to overcome these for case studies. Finally, the challenges are discussed, facing empirical studies in general and in this thesis in particular, in selecting research methods (Section 3.3).

3.1 Research Strategies in Empirical Software Engineering

Empirical research is 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 *descriptive* (finding the distribution of specific characteristics or attributes), *explanatory* (explaining why certain techniques are chosen), or *exploratory* (investigating parameters or doing a pre-study to decide whether all parameters of a study are foreseen). Software engineering is a cross-disciplinary subject area and is developing fast. In order to perform valid and reliable scientific research in software engineering, we have to understand the research methods, their purpose, limitations and when and how they can be applied [Wohlin00].

According to the literature on this subject [Creswell94] [Seaman99] [Wohlin00] [Creswell03], there are three types of research paradigms that have different approaches to empirical studies:

- *Qualitative research* is concerned with studying objects in their natural setting. Data used herein are usually words and pictures, not numbers. A qualitative researcher attempts to interpret a phenomenon based on explanations that people bring to it.
- *Quantitative research* is primarily concerned with quantifying a relationship or comparing two or more groups. The aim is to investigate a possible cause-effect relationship. A quantitative research is often performed through setting up controlled experiments or collecting data through surveys or case studies.
- *The mixed-method approach* is developed to compensate for limitations and biases in the aforementioned strategies, seeking convergence across other methods. Collecting data from multiple sources to address the same fact or phenomenon is also called triangulation. According to Seaman [Seaman99], a

combination of both qualitative and quantitative techniques is often more beneficial than either in isolation. Additionally, they should also be regarded as complementary rather than competitive. Earlier research [Basili96b] [Seaman99] describes how to combine qualitative and quantitative research methods.

An overview of empirical research approaches and examples of strategies for each is shown in Table 7, which relies on [Creswell03] [Mohagheghi04b].

Table 7. Overview of empirical research approaches [Creswell03] [Mohagheghi04b]

	Approaches		
	Quantitative	**Qualitative**	**Mixed methods**
Strategies	▪ Experimental design ▪ 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-centred ▪ Pluralistic

Table 7 shows how empirical research methods can be classified into different categories. The boundaries between the different research methods are not sharp. For example, case studies can combine both quantitative and qualitative studies.

In the paper by Zelkowitz and Wallace [Zelkowitz98] the authors have summarized 12 software engineering validation models. Furthermore, they have developed a taxonomy that describes these models according to the data collection methods: observational, historical, and controlled. However, we have only used research methods from one of the three major categorizations proposed by Zelkowitz and Wallace [Zelkowitz98], namely observational. In the observational category, we conducted different case studies in one company. Robson [Robson93] also presents three different types of investigations (strategy) that can be carried out. His categorization also includes case study and experiment as in [Zelkowitz98], but he also considers survey. The majority of our studies are based on a case study, but one is based on a survey. In the following two Sections, 3.1.1 and 3.1.2, a brief and general description is given of the two research methods used in this thesis. However, see Sections 4.3.1-4.3.4 for a more thorough description of how they were applied in our studies.

3.1.1 Survey Approach

Surveys are conducted when some piece of technology already has taken place [Pfleeger94], or when we want to explore past phenomena. They can also be used for opinion polls and market research, and the collected information can either be analysed quantitatively or qualitatively. Surveys can be seen as a snapshot of the current situation. The book by [Fowler88] writes about how to perform a valid and reliable survey, such as questionnaire design, sampling, and how to contact respondents. The key idea of *sampling* is to select some of the *elements* (e.g. a person) in a *population* (the total collection of elements which we want to make some decisions about) [Cooper03]. A valid sample is a representative subset of the study population [Conradi05], and the different types of sample design can be studied more thoroughly in [Cooper03] [Conradi05]. Even though surveys are most useful for studying numerous elements using a large sample size and extensive statistical analysis, we should always try to obtain the greatest amount of understanding from the fewest number of elements.

Surveys are used when the control of the independent and dependent variables is not possible, when the phenomenon of interest must be studied in its natural setting, and when this phenomenon occurs in the present or the past. Surveys are especially suited for answering questions about what, how much, and how many, as well as questions about how and why [Pinsonneault93]. Surveys are an empirical study often used in disciplines such as marketing, medicine, psychology and sociology. There is also a long tradition for using surveys to study organizational changes [Baumgartel59] [Neff66] [Kraut96].

In [Trochim08], several factors are considered when designing a survey. However, in the software engineering field, when conducting surveys the most troublesome parts are *selecting the sample frame* and *follow-up* of the prospective respondents. Selecting the sample frame can be difficult and time consuming. For example, if the survey is performed in the industry we first need to decide which companies go into our survey, and then we decide the projects. The result of this may be that the chosen sampling frame leaves out projects that are interesting and includes those that are not. Following-up on the respondents is also time and resource consuming, since we need to collect answers from the respondents.

The two most common data collection methods for surveys are through questionnaires and interviews [Wohlin00]. Questionnaires with mostly closed answer alternatives could be provided as either paper forms or in some electronic format, such as through email or web pages. One of the methods for data collection could be to let skilled interviewers fill in the questionnaires (by telephone or face-to-face meetings) instead of the respondents themselves. Wohlin et al. [Wohlin00] list some advantages with interview-driven surveys:

- Interview surveys achieve a higher response rate, compared to mail surveys.
- An interviewer decreases the number of inaccurate “do not know” answers, since the interviewer is available to answer questions about the questionnaire.
- The interviewer has the possibility to observe and ask questions.

However, the interviewer may introduce a bias for qualitative questions. The claimed disadvantages with surveys is time and cost, which depend on the size of the sample and the intentions of the investigation [Wohlin00].

Together with experimental research, survey research is a traditional approach that is supported by a rich social-science literature, describing how to design and administer it [Dybå01]. General introductions and guidelines for surveys are found in books like [Selnes99] [Wohlin00] [Cooper03].

3.1.2 Case Study Approach

A case study is conducted to investigate a phenomenon within a specific context and time interval. It is suitable for industrial evaluation of software engineering methods and tools because it can avoid typical scale-up problems (when you try to increase the scale from the laboratory to a real project) observed in small experiments [Kitchenham95]. Whereas formal experiments record the variables that are being manipulated, case studies collect information from the variables representing the typical situation. Case studies emphasize what is happening on a typical project: "research-in-the-typical". Since, formal experiments must be carefully controlled they are often small scaled: "research-in-the-small". Additionally, they also require appropriate levels of replication, as well as random assignment of subjects and objects. On the other hand, surveys try to capture what is happening in the population of respondents: "research-in-the-large". Generally, the most important aim of a case study is to *explain* the factors in a real-life context that are too complex for the survey or experimental approaches.

Yin [Yin03] defines a case study as:

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.

Furthermore, Yin [Yin03] has identified situations when case studies are more appropriate; "*when a **how** and **when** question is being asked about a contemporary phenomenon, over which the investigator has little or no control*". Case studies can also be used to evaluate the difference between two design methods. This means to determine "which is best" of the two methods [Yin94]. During the performance of a case study, a variety of different data collection procedures may be applied [Creswell94].

In software engineering, industrial case studies are rare due to several factors [Kitchenham95] [Wohlin00] [Mohagheghi04b]:

- *Confidentiality*: Many companies do not allow outsiders to access critical information, or publish the results. Some of the reasons could be confidentiality of results or the risk of intervening with the on-going project.
- *Longitudinal*: Performing a case study may need observation and collection of data over months or even years.

- *Planning and generalization*: Earlier literature [Kitchenham95] [Wohlin00] notes that case studies are easy to plan, but the results are difficult to generalize and even harder to interpret. There are issues that make the planning difficult, such as it takes time to get the necessary permissions, overcome the communications barrier, and understand the context. Hence, the results are more difficult to interpret and generalize due to the impact of the context.
- *Organizational changes*: A case study may take another turn than anticipated; projects may be stopped, or changes in the personnel or environment may happen that affect the data collection.

Performing a good case study involves the following steps [Kitchenham95]:
- Specify the research questions under test.
- Use state variables for project selection and data analysis.
- Establish a basis for comparisons.
- Plan case studies properly.
- Use appropriate presentation and analysis techniques to assess the results.

3.2 Measurement and Metrics

Software measurement is crucial to be able to control projects, products and processes. Hence, it is central in any empirical study, especially for benchmarking (collecting and analysing data for comparison), and for evaluating the effectiveness of specific software engineering technologies [Fenton00b].

Measurement and measure are defined as [Fenton96] [Wohlin00]:

***Measurement** is a mapping from the empirical world to the formal, relational world. Consequently, the term **measure** is the number or symbol assigned to an entity by this mapping in order to characterize an attribute.*

The term software *metrics* is either used to denote the activities in the field of measurement, or concretely in this thesis to denote a characterizing attribute which is measured according to a specified scale and by specified data collection and validation methods. For instance, the attribute software size is measured in source line of code, SLOC (the scale). Although the first book on software metrics was not published until 1976 [Gilb76], the history of software metrics dates back to the mid-1960s when the metric, Lines of Code, was used as the basis for measuring programming productivity and effort [Fenton00b].

While some attributes can be directly measured (e.g. number of defects found in a test), others are instead derived through other measures (that are directly measurable), and are called indirect measures (e.g. defect density in number of defects per SLOC). Measurement of an attribute can also be divided into objective and subjective measures; an objective measure is a measure where there is no judgment in the measurement value (e.g. delivery date). A subjective measure depends on both the object and the viewpoint, such as usability [Wohlin00].

Wohlin et al. [Wohlin00] have divided entities of interest that we wish to measure in software engineering into three classes:

- *Process*: describes which activities are needed to produce some software.
- *Product*: are the deliverables or documents which result from some activities.
- *Resources*: entities needed for a process activity, such as personnel, hardware or software.

Metrics are classified in five scale types: nominal, ordinal, interval, ratio and absolute scales. General definitions and statistical tests for each type are described in [Wohlin00] [Cooper03], for example.

Defining metrics and collecting related measures in an organization is a non-trivial task, considering the need for resources, time-consumption and cost. One approach that has been helpful for defining goals and collecting metrics in organizations is use of the GQM (Goal/Question/Metric) approach. GQM is developed by Victor Basili et al. and the Software Engineering Laboratory at the NASA Goddard Space Flight Center. More thorough information about GQM can be found in [Basili84] [Basili92] [Basili93].

3.2.1 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. Four categories of validity threats have been defined by Wohlin et al. [Wohlin00], assumed applicable outside formal experiments:

- *Conclusion validity – "right analysis"*: is concerned with the relationship between the treatment (refers to one particular value of one or more *independent* variables[3]) and the outcome. It is important to make sure that there is a statistical relationship (usually indicated by a low p-value[4]) of significance. Threats are related to issues such as statistical tests, and the reliability of measures.
- *Internal validity – "right data"*: a causal relationship between treatment and outcome, we must make sure that it is not due to factors we cannot control or measure. Threats are related to factors such as history, testing, and selection.
- *Construct validity – "right metrics"*: is concerned with the design of the study. We must ensure that the treatment reflects the cause and the outcome reflects the effect. Threats are related to issues such as interactions of different treatment, and hypothesis guessing.
- *External validity – "right respondents"*: is concerned with the generalization of results outside the scope of a study. Three common risk factors are: respondents (the subjects) are not representative for the population, environment (the context) is not representative, and time (the experiment is conducted at an inappropriate time).

[3] Independent variables are the development method, the experience of the personnel, tool support and the environment [Wohlin00].

[4] When a statistical test is conducted it is possible to calculate the lowest possible significance (often denoted p-value) [Wohlin00]. This p-value gives us the chance to reject the null hypothesis.

Different validity threats exhibit different priorities based on research method. Thus, for a case study, Yin [Yin03] identifies three tactics to improve validity:

- Use multiple sources in data collection and have key informants review the report in composition to improve construct validity.
- Perform pattern matching (comparing an 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.

Performing case studies in industry is valuable, since they allow us to evaluate methods in "real-life" contexts, to gather useful data for researchers, and to evaluate technology for researchers and practitioners.

3.3 Summary and the Challenges of this Thesis

Research methods and metrics were discussed in this chapter, and made us aware of the challenges faced when we plan a thesis like this. This section presents those research challenges facing empirical studies in general and in this thesis, especially in selecting research methods. The numeration of Research Challenges (RC) is continued from Section 2.6:

RC5. Defining research questions: What are the relevant research questions and how well are they defined? Sometimes the research question is well defined, making it easier to decide the research method. However, in most cases, the research question is emerging and so is the research method. In this thesis **RQ1** and **RQ2** were derived after a bottom-up analysis from the collected data, while **RQ3** was derived from a top-down analysis after reading existing literature on software reuse practice.

RC6. Choosing the most suitable research method: What research method should be "chosen" (e.g. in some cases the research method is given in advance due to the circumstances) to answer the research question(s)? The quantitative, qualitative or a combination of both (e.g. mixed-method research approaches) are briefly 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, in this thesis a mixed-method design is chosen that combines the results of a quantitative survey followed by a qualitative semi-structured interview, with quantitative analysis of industrial databases followed by a qualitative RCA on software change data. Sections 4.3.1 through 4.3.4 explain the research design for each of the individual studies that make up this thesis.

RC7. Collecting and analysing data: How should data be collected and analysed? The selected metrics and statistical tests are described in the papers **P1**-**P6** in Appendix A.

RC8. Validity: How valuable and valid are the discovered results? The relevant validity threats for each conducted study are presented and discussed in the papers (see Appendix A). In Section 6.6 the validity threats for all the studies are discussed.

4 Research Context and Design

This chapter presents the research focus (Section 4.1) and the company context in more detail (Section 4.2). Section 4.3 presents the overall research design and how it combines quantitative and qualitative studies. Additionally, a more detailed description of each study is also given. An overview of the study design is given (Section 4.4). Finally, the chapter summarizes how the research designs have impacted each other (Section 4.5).

The work of this thesis is divided into four main studies (see Table 1, Section 1.4). The details of the research design for each of these studies are discussed more thoroughly in Chapter 4, while the results of these studies are presented in Chapter 5.

4.1 Research Focus

This thesis is part of the SEVO project [SEVO04], and its four project goals are presented in Section 1.2. StatoilHydro ASA has cooperated with the SEVO project and has given us access to data for analysis and feedback. The company initiated its reuse strategy in 2003, and this strategy is now being propagated to other divisions in the company. Our main **RG** for this thesis from Section 1.3 was:

- *Investigate the advantages/disadvantages of systematic software reuse and the reasons behind it, by analysing software change data. Then, based on these insights, propose specific reuse guidelines (as an example of improvements) to StatoilHydro ASA, as well as general recommendations to software practitioners.*

The overall **RG** has been the guiding theme for the research in this thesis. Having an overall theme allowed us to adapt our research to the company preferences without deviating too far from our original directions. Given the results from our four main studies, we have broken the overall **RG** down into three explicit research questions which allow us to classify our findings. Thus, the following research questions were formulated together, with the reason why they are considered important:

- **RQ1: What is the relation between software changes and software reuse, by comparing the reused framework vs. software reusing it?**
    - Software changes are important because they account for a major part of the costs of the software. By characterizing and explaining the software

changes to the reused framework vs. software reusing it, possible management strategies to StatoilHydro ASA can be suggested, depending on which type(s) of changes are more prevalent.

- **RQ2: How do the reused framework and software reusing it evolve over time?**
    - Software reuse is expected to improve software productivity and quality, reduce cost and time-to market, standardization, dissemination of best-practice etc. Although many empirical studies have investigated the *snapshot* aspects of changes[5], few have explored them from the *longitudinal* aspect (i.e. how the changes vary over time). By comparing the software change profile for the reused framework and software reusing it, we can show whether or not defect density, CR density and change density of the reused framework and software reusing it improve over time. This is important for the company for assigning development resources. However, if the reused framework vs. software reusing it has several users and/or rather intense testing, it can lead to high defect density, CR density and change density. Thus, it does not necessarily mean that the reused framework vs. software reusing it have a poorer "quality" (see Section 2.2). Hence, defect density, CR density and change density cannot be used as a standard measure for quality.
- **RQ3: What improvements can be made towards the actual reuse practice at StatoilHydro ASA?**
    - Developing for reuse is risky, like any other investment for an uncertain future. Therefore, we need to investigate the (critical) success factors of past and ongoing reuse programs. This research question emerged from two factors, namely combining results from **RQ1** and **RQ2**, and after exploring some of the results from **P1**.

In the beginning it was decided to study attributes like productivity (person hours/NSLOC), time-to-market (visualize planned and real progress and cost at a defined point in time) and test maturity (number of passed test cases vs. number of planned test cases). However, when we started to collect data from the three industrial software systems, these attributes had to be eliminated. This was due to lack of complete information in the data material. Hence, the research questions and metrics had to be redefined. See Section 4.2.5 for the main metrics we identified for studying trouble reports, change requests and software change data related to the NSLOC.

Table 1 (Section 1.4) contains a short description related to *focus* and *research method* of the studies we have performed, and each study is elaborated in Section 4.3.

[5] i.e. distribution of different kinds of changes, or the distribution of effort spent on performing different kinds of changes.

4.2 The StatoilHydro ASA Context

4.2.1 The Investigated Company

StatoilHydro ASA has a total of about 31,000 employees, with its headquarters in Norway and branches in 40 countries. The IT department of the company is responsible for developing and delivering domain-specific software, to give key business areas better flexibility and efficiency in their regular operations. It is also responsible for the operation and support of mass IT systems. This department consists of approximately 100 developers, located mainly in Norway. In addition, StatoilHydro ASA subcontracts a great deal of software development, maintenance and operations to consulting (software) companies, and over 1000 consultants are regularly engaged in such activities.

4.2.2 The Investigated Software Systems

Three systems have been investigated in this thesis. One is a reused framework called JEF. The remaining two, which reuse JEF, are applications called DCF and S&A.

The company initiated their reuse strategy in 2003 with pre-studies. However, DCF and S&A were not initially designed and implemented for future reuse, while JEF was developed for reuse and is based on J2EE (Java 2 Enterprise Edition). It is a Java technical framework for developing Enterprise Applications. Thus, the framework is called the “JEF framework” and consists of seven separate components or classes (see Section 4.2.3 for a more thorough description of these components). The decision of which components to include in the framework was based on a performed technical analysis related to the business needs in the company, which resulted in the need of the following sub-components: (1) GUI components, (2) business logic components, and (3) security components. Hence, these components were included in the first release of the framework. JEF has been generalized for reuse by an earlier application, namely PDM (Physical Deal Maintenance, see Figure 6). After the framework was reused by DCF and S&A, other components were included. The components in this framework are built from a combination of COTS (Commercial-Off-the-Shelf) components, OSS (Open Source System) components, and some in-house built code. The latest release of JEF components contained a total of 20348 Non-commented Source Lines of Code (NSLOC), and can either be applied separately or together when developing applications. Table 8 shows the size and release date of the three JEF releases (excluding third-party components).

DCF is meant to replace the current handling of cargo files, which are physical folders that contain printouts of documents that pertain to a particular cargo or contract. A “cargo file” is a container for working documents that are related to a cargo or contract that is used by all parties in the oil sales, trading, and supply strategy plan of the company. There are three releases of the DCF application. Table 8 gives an overview of the size and release date of the three DCF releases (excluding the code of JEF and other third-party components).

S&A is an application for providing more efficient and controllable business processes for lift and cargo planning. **Lift planning** is based on a "lifting program" to generate an overview of the cargoes that are scheduled to be lifted in the future. The **cargo planning** and shipment covers activities to accomplish such lifting. The current trading system ("SPORT") is not able to handle complex agreements (i.e. mixing of oil qualities within the same shipment), or automating the transfer and entry of related data (currently often manual). The main goal of the S&A application is to replace some of the current processes/systems, as well as to offer some new functionality. The S&A application also has three releases. Table 8 gives an overview of the size and release date of these releases (excluding the third party components).

Table 8. Size and release date of the three systems

Systems	Release 1		Release 2		Release 3	
	Date	Size (NSLOC)	Date	Size (NSLOC)	Date	Size (NSLOC)
JEF	14 June 2005	16875	9 Sept. 2005	18599	8 Nov. 2005	20348
DCF	1 Aug. 2005	20702	14 Nov.2005	21459	8 May 2006	25079
S&A	2 May 2006	29957	6 Feb.2007	50879	12 Dec.2007	64319

From Table 8 we can see that the framework and the applications are growing. JEF consists of seven components (see Section 4.2.3). These are being used in PDM and reused in DCF and S&A (see Figure 6). However, DCF and S&A are not being used in any other applications. JEF is a framework that is reused in DCF and S&A and in other projects "as-is". This is how we can say that JEF is developed *for* reuse, and DCF and S&A are developed *with* reuse. JEF, DCF, and S&A will grow in size because when the clients use the applications they will make some changes to them, which will also require changes to the framework. For instance, adding new functionality to the reused framework and software reusing it will result in growth for JEF, DCF, and S&A. Another explanation of the growth of the framework and the applications is that when a defect is found in Release 1 the fixes will be included in Release 2, etc. Thus, the framework and the application will grow.

JEF Release 1 was finished around June 2005, and PDM in the summer 2005 was the first application to use the JEF framework (Release 1). In this period, some weaknesses in the framework were discovered. These changes were then incorporated into JEF, ending early September 2005. Then, Release 2 of the JEF framework was delivered. The DCF application reused Release 2 of the JEF framework during late summer and autumn 2005. After DCF reused the JEF framework, some more minor changes were made to the framework, which were finished by early November 2005. Then, Release 3 of the JEF framework was deployed. The second application, S&A, reused Release 3 of the JEF framework, and was developed during early 2006. The relation between JEF and applications using/reusing it are shown in Figure 6.

The company uses the same test team and has the same test coverage for both the reused framework and software reusing it. For instance, for unit testing, 85% of the code lines were executed by unit tests to ensure that the code worked as expected. However, detailed investigation of software testing will be the topic of Future Work (see Chapter 7). We have not included defects in the PDM application other than those in JEF in our study, because PDM was the first application to *use* JEF, not *reuse* it (like DCF and S&A).

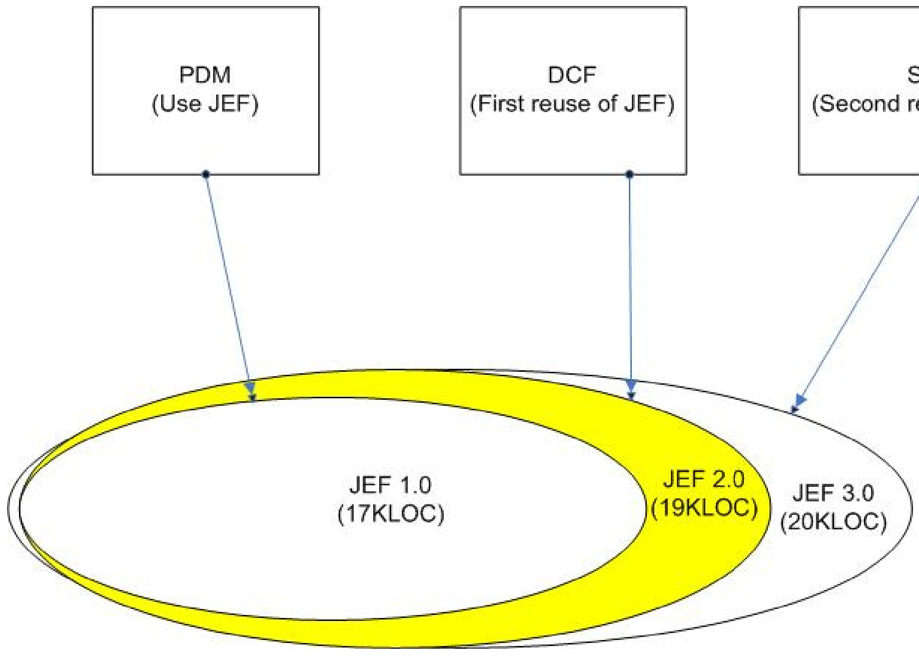

Figure 6. The relation between JEF, DCF and S&A

4.2.3 The Reused Framework: JEF

Figure 7 depicts the JEF components, and the components coloured in green are the ones reused “as-is” in DCF and S&A and other business applications. The one component coloured in grey, namely *JEFSessionManagement*, is just used internally in the framework.

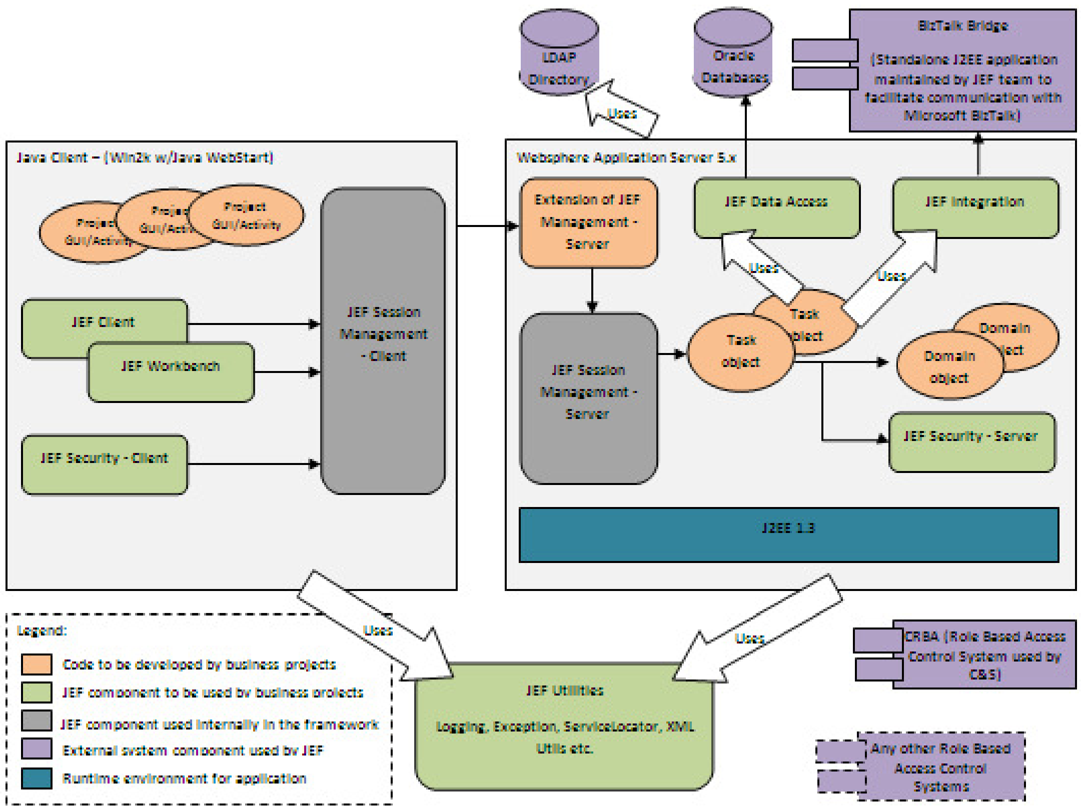
LDAP Directory
Oracle Databases
BizTalk Bridge
(Standalone J2EE application maintained by JEF team to facilitate communication with Microsoft BizTalk)
Uses
Java Client – (Win2k w/Java WebStart)
Project GUI/Activity
Project GUI/Activity
Project GUI/Activity
JEF Client
JEF Workbench
JEF Security - Client
JEF Session Management - Client
Websphere Application Server 5.x
Extension of JEF Management - Server
JEF Data Access
JEF Integration
Uses
Uses
Task object
Task object
Domain object
Domain object
JEF Session Management - Server
JEF Security - Server
J2EE 1.3
Legend:
Code to be developed by business projects
JEF component to be used by business projects
JEF component used internally in the framework
External system component used by JEF
Runtime environment for application
Uses
Uses
JEF Utilities
Logging, Exception, ServiceLocator, XML Utils etc.
CRBA (Role Based Access Control System used by C&S)
Any other Role Based Access Control Systems

Figure 7. The JEF components

The following is a brief description of the seven components in JEF from Figure 7:

- *JEFClient* (8885 NSLOC): a large class library providing functionality (e.g. binding between data objects and the content in GUI), so that other applications using this component do not need to include low level client code.
- *JEFWorkbench* (4748 NSLOC): provides features like authentication, authorization, navigation, preferences, plug-ins, and much more. This component enables login for users and presents the activities authorized for each user.
- *JEFSecurity* (2374 NSLOC): provides both authentication and authorization services for running code on the client and on the server.
- *JEFUtil* (1647 NSLOC): provides different utilities such as service locators (provides communication between client and server), and xml utils (converts xml definitions into data objects and vice versa).
- *JEFSessionManagement* (1468 NSLOC): provides communication between the client and the server. The component can support any protocol supported by JAVA (e.g. HTTP/HTTPS). This component, however, is used internally in the framework.
- *JEFIntegration* (958 NSLOC): provides communication between application components, information systems and other systems/applications in the overall architecture.

- *JEFDataAccess* (268 NSLOC): provides data access to the developers, so they can use a common pattern when creating Java Data Objects (information objects we wish to persist).

JEF is designed on the basis of a technical architecture for all J2EE systems in the company. This architecture has four logical layers, and the following presentation is from top to bottom:

(1) Presentation: responsible for displaying information to the end-user and to interpret end-user input. JEF components on this layer are JEFWorkbench, JEFSecurity and JEFClient.

(2) Process: provides support for the intended tasks of the software, and configures the domain objects. JEF components on this layer are JEFSessionManagement and JEFIntergration.

(3) Domain: responsible for representing the concepts of the business, and information about the business and business rules. This layer is the heart of the system. JEF component on this layer is JEFDataAccess.

(4) Infrastructure: provides generic technical services, such as transactions, messaging, and persistence. JEF component on this layer is JEFUtil.

4.2.4 Development Environment and Tools

The development technology to implement the three software systems is J2EE (Java 2 Enterprise Edition), and SPRING class framework (OSS). The programming language is Java. The Rational ClearQuest and Rational ClearCase tools are used for SCM.

To handle changes in requirements or implemented artefacts, Change Requests (CRs) are written (by test manager or developers) and stored in the Rational ClearQuest tool. Examples of change requests are: add, modify or delete functionalities; solve a problem with major design impact; or adapt to changes from e.g. JEF component interfaces. Failures detected during integration/system testing and all field use are handled by the trouble reporting process and stored in the Rational ClearQuest tool (as defects).

The project leader or test managers distribute the change requests and trouble reports (from Rational ClearQuest) among the developers. The developers then access the source files in the version control system, i.e. Rational ClearCase, to make the necessary changes. When implementing the changes the developers adhere to the following steps:

- They check-out the files corresponding to the specific change request or trouble report.
- They implement changes on the checked-out files, with possible locking of the branch they are working on.
- They give the file a change description, a more thorough description, which is an elaboration of what changes they have made and a time and date (timestamp).
- In the end, they check-in the files back into Rational ClearCase.

Figure 8, an adaptation after [Mohagheghi04a], shows the phases and states in CR and TR handling processes. CCB stands for the *Change Control Board* (usually found in

SCM systems), who is responsible for approval or rejection of a CR or TR. The project leader in StatoilHydro ASA constitutes the CCB in this context.

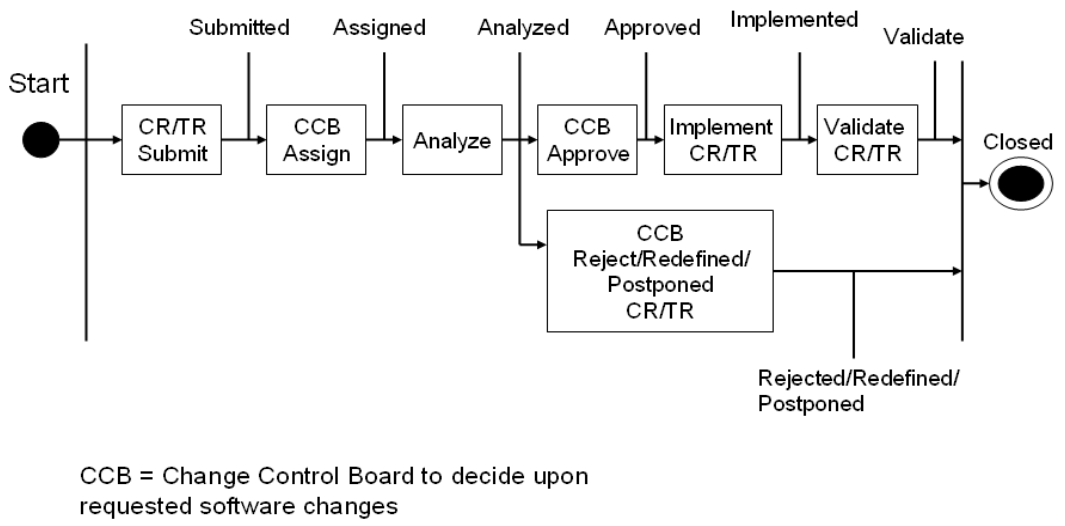
Start
Submitted
Assigned
Analyzed
Approved
Implemented
Validate
CR/TR Submit
CCB Assign
Analyze
CCB Approve
Implement CR/TR
Validate CR/TR
Closed
CCB Reject/Redefined/ Postponed CR/TR
Rejected/Redefined/ Postponed
CCB = Change Control Board to decide upon requested software changes

Figure 8. Software change process [Mohagheghi04a]

4.2.5 Data Collection and Metrics

After we were given access to the company data, we started to plan and collect the necessary data for conducting our empirical studies. The employed data collection methods were: survey with semi-structured interviews, and case studies where we (in some cases) performed a qualitative RCA. However, due to the nature of the industrial data, some of our research was based on a bottom-up data collection. That is, we explored the data material prior to formulating our research questions (see Section 4.3 for a thorough description). Prior research [Basili94] claims that the data collection approach should proceed in a top-down rather than a bottom-up fashion, for instance by employing GQM to define relevant metrics. However, Mohagheghi and Conradi [Mohagheghi04d] have given some reasons why bottom-up studies are useful:

- There exists a "*gap between the state of the art (best theories) and the state-of-the-practice (current practices)*" [Mohagheghi04d, p.64]. Therefore, most data gathered in company repositories are not collected according to the GQM paradigm.
- Several projects have been going on for a while without specifying improvement programs in the beginning, and may later want to start one. The companies want to "*assess and analyse the usefulness of the data that have already been collected and relate the data to the goals (reverse GQM)*" [Mohagheghi04d, p. 64].
- Even though a company has a measurement program with defined goals and metrics, "*these programs need to be improved from bottom-up studies*" [Mohagheghi04d, p.65].

Studying industrial data repositories can either be an exploratory study (identifying relations and trends in data material) or a follow-up study (confirmatory) to validate other or newer theories than those originally underlying the collected data [Mohagheghi04b].

This work has collected and analysed TRs, CRs, and changes to the code for the reused framework and software reusing it:

- CRs and TRs stored in Rational ClearQuest were analysed consequently in **P2** and **P5**.
    - The metrics that were used **directly** from the data in the change requests and trouble reports were the reported *description*, *classification, severity and priority*.
    - The relation between TRs and CRs was analysed in **P3**.
- Software changes to the source code were extracted from Rational ClearCase, and these data were analysed in **P4** and in **P6**.
    - TRs and CRs can be rejected, redefined or postponed, and both can lead to changes in different project reports and documents. It is important to investigate the actual changes made to the software system by exploring the source code.
    - The metrics that were used **directly** from the change data were the reported *filename* (with all of its *version numbers* and *dates*), *change description*, *code size* (NSLOC), *release number,* and *location of the change (i.e. JEF, DCF and S&A)*.
- The *number* of detected software changes of the overall JEF, DCF and S&A is a **direct** metric attained simply by counting the number of software changes of a certain type or for a certain system part etc.
- *Size* (NSLOC) of the overall JEF, DCF and S&A (a **direct** metric).
    - The size and number of defects were used to calculate the **indirect** metric defect density in **P3** and in **P5**.
    - The size and number of change requests were used to calculate the **indirect** metric change request density in **P3**.
    - The size and number of changes to source code were used to calculate the **indirect** metric change density in **P6**.

The collected TRs and CRs have missing or incomplete data regarding more fine-grained component information for DCF, i.e. we could not tell which concrete components in DCF have been affected by code changes. It is only for JEF and S&A that we have complete data on the component level. In paper **P6**, we investigated the change density related to the source code for JEF, DCF and S&A. However, we could not use component level data, since we were not able to trace back to the actual components affected by a change. This was due to the interface between Rational ClearQuest and Rational ClearCase. Each change performed on the source code in Rational ClearCase could be traced back to the CR submitted in the Rational ClearQuest. However, on several occasions we had problems tracing back the changes to the corresponding CR, and the actual components that were affected. For instance, the field "component" in Rational ClearQuest was incomplete for DCF, and in other cases we got an error message saying that the CR could not be identified. The configuration manager in StatoilHydro ASA did not know the reason for such error messages. Hence, we had to investigate CRs, TRs and changes to the source code for all three software systems, in the cases where we had to compare them. Our main motivation for **P3** and **P6** was to discover whether JEF, DCF and S&A become more stable over time (by investigating how the change request density in **P3** and the change density in **P6** evolved over successive releases). For both **P3** and **P6**, we could not conclude whether the observations were statistically significant or not, since we have no fine-grained data for the inside parts of these three software systems.

4.3 Research Approach and Design

The research in this thesis has combined qualitative studies of practice and processes, with quantitative studies of data collected from the company's repositories. We have further combined the results to propose software reuse improvements. The rationale for combining these different types of studies has been:

- Investigating industrial projects gives us the possibility to collect and analyse data, such as CRs, TRs, software changes to the source code, etc. Therefore, it is useful to take advantage of all available data.
- The results of our studies mostly confirmed other studies; i.e. triangulation of data, mainly by using qualitative methods.

Selecting research questions and research methods for this thesis has been both a top-down and bottom-up approach:

- The questionnaire in **P1** is based on previous work in the field [Li04].
- Most of our research questions stem from a mixture of literature studies in a top-down fashion, and exploratory work on available data sets and company practices, in a bottom-up fashion.

There have been three phases of this work, as shown in Figure 2 and Table 1 (see Section 1.4):

- **Phase 1 (Study 1)**: Consisted mainly of a quantitative survey followed by qualitative semi-structured interviews of developers' views on software reuse. This phase has followed a top-down approach.
- **Phase 2 (Study 2 and 3)**: Identified mainly by quantitative studies on change requests and trouble reports, but in **P5** a qualitative RCA was performed. This phase started with a top-down confirmatory approach and continued with more bottom-up explorative studies.
- **Phase 3 (Study 4)**: The results were mainly obtained from quantitative studies, but we also performed a qualitative RCA on change data related to the source code. This phase, just like phase 2, combined a top-down and a bottom-up approach.

Sections 4.3.1 through 4.3.4 explain the research design for each of the individual studies that make up this thesis. As mentioned in Section 4.1 we have broken the overall **RG** into three main research questions (see Section 1.3), and the individual studies are discussed according to these research questions. The contributions for each of these studies are presented in Table 10, and elaborated on in Chapter 5.

4.3.1 Study 1: Survey on Developers' Views on Software Reuse (Paper P1)

The goal for Study 1 was to investigate the opinions of developers on software reuse, related to five main areas: benefits of reuse, factors contributing towards reuse, possible relations between reuse and increased rework, component understanding, and quality attribute specification. In order to achieve this goal and to answer **RQ3**, we chose to perform a survey followed by qualitative semi-structured interviews (see Section 3.1.1 for a more through description of the survey). We chose this research method since it was clear that the information we sought could only be obtained directly from respondents rather than from the accumulated technical data. The questionnaire used in this study was based on previous literature (top-down process).

The developers that participated in the survey currently work with the DCF and S&A projects, reusing the JEF components developed by the JEF Team. Also, some of these developers are part of the JEF Team; that is, they both develop and reuse the JEF components. In total, there are 16 developers working with the DCF project, the S&A project and the JEF Team at Statoil ASA in Stavanger, Trondheim and Oslo. We asked all these developers to participate in the survey, and received 16 out of 16 completed questionnaires.

The developers answered the questionnaires separately, and they were filled out by hand. After the developers had completed the questionnaire we performed short semi-structured, one-on-one interviews with each of the developers for 10-15 minutes. This was done to provide support for possible misunderstandings in answering the questionnaire, as well as obtaining more thorough, qualitative information around the issues presented in the questionnaire.

The following are the research questions for Study 1, derived from **P1**:

- **RQ.S1.a:** *What are the key benefits of reuse?*
- **RQ.S1.b:** *Which factors contribute to facilitate reuse?*
- **RQ.S1.c:** *Does reuse increase rework?*
- **RQ.S1.d:** *Do developers have sufficient information to understand the relevant components? If the answer is no, how can they solve this problem?*
- **RQ.S1.e:** *Do developers trust the relevant quality specification of the components? If the answer is no, how can they solve this problem?*

Validity comment. Most of the questions in the questionnaire used in the survey have their origin from the research literature. Further, through pre-testing among local colleagues, most of the questions were refined additionally. Also, terms that may be unfamiliar to the respondents were defined in the questionnaire handout.

4.3.2 Study 2: Analysing Change Requests (Papers P2, P3)

In order to explore the profile of CRs (to answer **RQ1**), as well as to see how change density for the reused framework vs. software reusing it evolves over several releases (to answer **RQ2**), we chose to perform a case study, i.e. *data mining* (see Section 3.1.2 for a more thorough description of case study). We chose this research method (including study 3 and study 4) for these data since they were given to us by the company and are of a longitudinal nature.

This study investigates change requests (CRs) for JEF in **P2**, and JEF vs. DCF in **P3**. CRs related to origin (i.e. distribution of CRs over perfective, adaptive and preventive changes), priority level, and relation to component size for JEF were studied in the former paper. How change request density (number of CRs/KNSLOC) evolves over time for JEF vs. DCF were studied in the latter paper.

The goal was to improve the knowledge about CRs. Studying the change requests' distribution and change request density is important to discover where the majority of the effort related to CRs is being spent in StatoilHydro ASA, as well as to discover if JEF and/or DCF becomes more stable over time (see discussion in Section 4.2.5). Even though **P3** also investigated defect density, a more thorough description of the research questions related to defect density is presented in Section 4.3.3, and only research questions related to change request density are presented here.

Study 2 is a quantitative study based on data mining, and the change requests were collected from Rational ClearQuest for JEF and DCF. This investigation has been both a top-down and a bottom-up process. We have used related work (described in paper **P2** and **P3**, see Appendix A) and the CRs collected from the company to formulate our research questions.

The following are the research questions for Study 2, derived from **P2** and **P3**:

- **RQ.S2.a:** *How is the distribution of CRs over perfective, adaptive and preventive changes?*
- **RQ.S2.b:** *What is the relation between the customer priority and the developers' priority on CRs?*
- **RQ.S2.c:** *What is the relation between component size and the number of CRs?*
- **RQ.S2.d:** *What is the distribution of CRs over priority levels given by the developers?*
- **RQ.S2.e:** *How does the change density for JEF vs. DCF evolve over several releases?*

Validity comment. The change categories we have used to classify CRs in **P2** and the metric change request density in **P3** are thoroughly described and used in literature. All of the change requests in **P2** have been classified manually by us. To enhance the internal validity of the data we have classified all the CRs separately, and then cross-validated the results.

4.3.3 Study 3: Analysing Trouble Reports (Papers P3, P5)

This study investigates trouble reports (TRs) for JEF vs. DCF in **P3**, and has in **P5** included an extra application reusing JEF, namely S&A.

The goal for this study has been two-fold. The first goal was to get a deeper insight into how defect density (number of defects/KNSLOC) evolves over time, and the relation between defect density and change request density for JEF vs. DCF (change request density described in Section 4.3.2), to answer **RQ2**. The second goal was to compare the defect profile (in terms of defect density, density of specific defect types, and the severities/impacts of defects) for the reused framework (JEF) vs. software reusing it (DCF and S&A), to answer **RQ1**. By viewing both goals together, it made it possible for us to characterize and verify possible reuse benefits (to answer **RQ3**). In order to achieve our two goals and to answer our overall research questions, we chose to perform a case study, i.e. *data mining*.

Study 3 is mainly a quantitative study based on data mining, but in **P5** a qualitative RCA was also performed. The trouble reports were collected from Rational ClearQuest for JEF, DCF and S&A. As in study 2, this investigation has also been a combined top-down and a bottom-up approach. Related work (described in papers **P3** and **P5**, see Appendix A) and the collected TRs from the company have been used to formulate our research questions.

The following are the research questions for Study 3. These are derived from **P3** and **P5**:

- **RQ.S3.a:** *How does the defect density for JEF vs. DCF evolve over time?*
- **RQ.S3.b:** *What is the relation between change density and defect density for JEF vs. DCF?*
- **RQ.S3.c:** *What is the overall defect density of JEF vs. DCF and S&A?*
- **RQ.S3.d:** *What is the density of specific types of defects in JEF vs. DCF and S&A?*
- **RQ.S3.e:** *What are the severities and the most severe defects in JEF vs. DCF and S&A?*
- **RQ.S3.f:** *What impacts on the client do defects in JEF vs. DCF and S&A have?*

Validity comment. The metric defect density in both **P3** and **P5** is thoroughly described and used in the literature. Additionally, all of the trouble reports used in **P5** were classified manually by us, using ODC. Thus, to enhance the internal validity of the data we classified all the defects separately, and then cross-validated the results.

4.3.4 Study 4: Analysing Change Data Related to the Source Code (Papers P4, P6)

This study investigates software change data related to the source code for JEF vs. DCF in **P4**, and in **P6** has included an extra application reusing JEF, namely S&A. Since both TRs and CRs can be rejected, redefined or postponed, and both can lead to changes in different project reports and documents, we decided to study the actual changes made to the source code.

The goal has been to study the change profile (e.g. frequency, change type and change profile over time) for the reused framework vs. software reusing it (to answer **RQ1** and **RQ2**), and to investigate the maintenance benefits and challenges of software reuse (to answer **RQ3**). In order to achieve our goal and to answer our overall research questions, we chose to perform a case study, i.e. *data mining*.

Study 4 is similar to study 3, when it comes to research design (i.e. quantitative), research approach (i.e. top-down and bottom-up), and formulation of research questions (see **P4** and **P6** in Appendix A). The exceptions are that a qualitative RCA was used in both **P4** and **P6**, and that the change data were collected from Rational ClearCase for JEF, DCF and S&A.

The following are the research questions for Study 4, derived from **P4** and **P6**:

- **RQ.S4.a:** *Does the distribution of change types vary for different development characteristics (i.e. designing for reuse and before/after refactoring)?*
- **RQ.S4.b:** *What change types are the longest for different development characteristics?*
- **RQ.S4.c:** *How localized are the effects of different types of changes for different development characteristics?*
- **RQ.S4.d:** *Whether the reused framework experienced fewer or more changes than applications reusing it, and the reasons for the differences or similarities?*
- **RQ.S4.e:** *Whether the reused framework experienced the same profile of changes over time with the applications reusing it, and the reasons for the differences or similarities?*

Validity comment. The change density metric in **P6** is thoroughly described and used in the literature. All of the software changes to the source code (see Figure 4 for definition) investigated in both **P4** and **P6** have been classified manually by us. We have classified all the source code changes separately, and then cross-validated the results. This was to enhance the internal validity of the data.

4.4 An Overview of the Studies

Table 9 gives an overview of how the sub-research questions for each individual study relates to the main research questions in this thesis.

Table 9. Relation between main and sub-research questions

Research questions	**Sub-research questions**
RQ1	**RQ.S2.a-RQ.S2.d,** **RQ.S3.c-RQ.S3.f,** **RQS4.a-RQ.S4.d**
RQ2	**RQ.S2.e,** **RQ.S3.a-RQ.S3.b,** **RQ.S4.e**
RQ3	**RQ.S1.a-RQ.S1.e,** and combining results from RQ1 and RQ2.

Table 10 gives an overview of the Research Questions (RQ) and their relations to the studies, together with the phases, type of studies and contributions.

Table 10. Types of studies and their relations to phases, RQ, papers and contributions

Phase	Studies	Study description	Research design	RQ1	RQ2	RQ3	Paper	Contribution
1	Study 1	Study on developers' views on software reuse	Quantitative, with qualitative semi-structured interviews.			✓	P1	C4
2	Study 2	Study of CRs	Quantitative and exploratory study of data repository.	✓	✓		P2, P3	C1, C3
	Study 3	Study of TRs	Quantitative/ qualitative RCA in P5 and exploratory study of data repository.	✓	✓	✓	P3, P5	C2, C3, C4
3	Study 4	Study of change data related to the source code	Quantitative/ qualitative RCA and exploratory study of data repository.	✓	✓	✓	P4, P6	C1, C3, C4

4.5 Summary

The research context was presented in Sections 4.1-4.2, while the research approach and design of the individual four studies that makes up this thesis was presented in Sections 4.3.1-4.3.4. Further, an overview of the studies was given in Section 4.4. This thesis has combined qualitative studies of practice and processes, with quantitative studies of data collected from the company's software repositories. We first conducted a survey which was followed by semi-structured interviews. This gave us insight into the developers' views on software reuse. This made the basis for our further work, which was based on case studies (i.e. *data mining* of the company repositories). The reason for choosing case studies was the available data, i.e. CRs, TRs and change data related to the source code stored in Rational ClearQuest and Rational ClearCase. After the quantitative data analysis of the collected data, we performed a qualitative Root Cause Analysis (RCA) for **P4**, **P5** and **P6**, by interviewing a senior developer who was familiar with development of all JEF, DCF and S&A. We first showed him the results of our data analysis (to avoid a possible threat to validities of our results, we did not inform him of our research questions), and then we asked him to interpret the results. Hence, the combined research methods have helped us answering our three main research questions. The next chapter presents the main results of papers **P1**-**P6** in light of the four main studies.

5 Results

This chapter summarizes the results of the research for each of the four studies (Sections 5.1-5.4). The results are reported in more detail in the papers in Appendix A. Further, the contributions made by this thesis are presented (Section 5.5). Finally, the chapter sums up the main findings described in Chapter 4 and Chapter 5 (Section 5.6).

Sections 5.1-5.4 are furthermore divided into three: first we give a brief introduction of the *abstract* of the papers related to each of the four studies in this thesis, then we present the *results* for the papers, and in the end an overview is given of the *contributions* resulting from each of the four studies.

5.1 Study 1: Survey of Developers' Views on Software Reuse (Paper P1)

Study 1 was carried out by doing a quantitative survey with qualitative, semi-structured interviews on developers' views on software reuse. The results have been presented in one paper, namely **P1**. Our work in this study answers the research question:

- **RQ3:** W*hat improvements can be made towards the actual reuse practice at StatoilHydro ASA?*

P1. An Empirical study of Developers' Views on Software Reuse in Statoil ASA
Abstract for P1. In this article, we describe the results from our survey in the IT-department of a large Oil and Gas company in Norway (StatoilHydro ASA), in order to characterize developers' views on software reuse. We have used a survey followed by semi-structured interviews, investigating software reuse in relation to requirements (re)negotiation, value of component information repository, component understanding, and quality attribute specifications. All 16 developers participated in the survey and filled in the questionnaire based on their experience and views on software reuse. Our study focused on components built and reused in-house.

Results for P1. Our main results from this study show that the main *reuse benefits* from the developers' viewpoints include lower costs, shorter development time, higher quality of the components developed for reuse, and a standardized architecture. Factors *facilitating reuse* were: formal processes for general software development (implicit positive effect), dynamic and interactive JEF documents, and a shared information and experience repository for JEF components (e.g. for storing information regarding JEF components, rather than the components themselves). The qualitative reasons given include easier information sharing, easier learning, improved quality of documentation, and a better overview of the documentation and functionality, as well as of existing problems and troubleshooting. Finally, our results revealed that component understanding was generally sufficient, but documentation could be improved.

Contributions of Study 1:
The contribution of this study was to show the benefits of reuse and factors contributing to successful software reuse in the company. This study supports the main contribution **C4:** *Identification of possible software reuse improvements.*

5.2 Study 2: Analysing Change Requests (Papers P2, P3)

Study 2 was conducted quantitatively by data mining. The results have been published in papers **P2** and **P3**. Even though the latter paper also investigated defect density, those results are presented in Section 5.3. Only the results related to change request density are presented here. Our work in this study answers the research questions:

- **RQ1** (by **P2**): *What is the relation between software changes*[6] *and software reuse, by comparing the reused framework vs. software reusing it?*
- **RQ2** (by **P3**): *How do the reused framework and software reusing it evolve over time?*

P2. An Empirical Study of Software Changes in Statoil ASA – Origin, Priority Level and Relation to Component Size
Abstract for P2. This paper describes the results of analysing change requests from 4 releases of a set of components developed for reuse by StatoilHydro ASA. These components total 20348 SLOC (Source Lines of Code)[7], and have been programmed in Java. Change requests in our study covered any change in the requirements. We have investigated the distribution of change requests over the categories perfective, adaptive and preventive changes that characterize aspects of software maintenance and evolution. In total there were 208 combined perfective, adaptive and preventive changes. The corrective changes (223 in total) were excluded in this paper since they will be analysed in future work. We have also investigated the relation between customers' and developers' priorities on change requests, distribution of change requests over priority levels given by developers, and the relation between component size and number of change requests.

Results for P2. Developers' efforts for the reused framework were related to *perfective* (59%), *adaptive* (27%) and *preventive* (14%). On customers' vs. developers' priorities on change requests we found no significant differences between their priorities of change requests. However, the data show that there was a difference for critical change requests, though this was not statistically significant. The data trend was that the customer assigned more change requests on the *critical* level than the developers, while

[6] As mentioned in Section 1.2, software changes here refers to all changes done on software systems, i.e. defect changes and non-defect changes (see Figure 1).
[7] Even though we in **P2** have written SLOC it is actually NSLOC.

developers assigned more on the *high* level than customers. This is due to the developers having downgraded critical ones to high priority. Furthermore, larger components had more change requests than expected. This may not be a surprising result, but verifying this was important to StatoilHydro ASA in order to show where the majority of change requests actually occured. Finally, we found that the larger components had more serious (critical and high) change requests than the smaller ones, and that the smaller components did not have critical change requests at all.

P3. A Case Study of Defect Density and Change Density and their Progress over Time

Abstract for P3. We have performed an empirical case study, investigating defect density and change request density of a reused framework compared with one application reusing it over time at StatoilHydro ASA. The framework, called JEF, consists of seven components grouped together, and the application, called DCF, reuses the framework, without modifications to the framework. We analysed all trouble reports and change requests from three releases of both. Change requests in our study covered any changes (not correcting defects) in the requirements, while trouble reports covered any reported defects. Additionally, we have investigated the relation between defect density and change request density both for the reused JEF framework and the application.

Results for P3. The main findings showed us that the JEF framework had *higher* change request density in the first release, but *lower* change request density than the DCF application over the successive releases. For the DCF application, on the other hand, a slow *increase* in change request density appeared.

Contributions of Study 2:
In paper **P2** the contribution was to characterize and explain the changes to components, which is an indication as to which kind of components require more effort and resources in being managed in StatoilHydro ASA. This was related to the main contribution **C1:** *Identification of differences/similarities of the change profile for the reused framework vs. software reusing it*. In paper **P3** the contribution was to gain more insight about software evolution by exploring how change request density evolved over time for JEF and DCF. This was related to the main contribution **C3:** *Description of the software change lifecycle for the reused framework vs. software reusing it.*

5.3 Study 3: Analysing Trouble Reports (Papers P3, P5)

Study 3 was, like Study 2, conducted in a quantitative way based on data mining, but in **P5** a qualitative RCA was also performed. The findings have been presented in papers **P3** and **P5**. Our work in this study answers the research questions:

- **RQ1** (by **P5**): *What is the relation between software changes and software reuse, by comparing the reused framework vs. software reusing it?*
- **RQ2** (by **P3**): *How do the reused framework and software reusing it evolve over time?*
- **RQ3** (by **P5**): *What improvements can be made towards the actual reuse practice at StatoilHydro ASA?*

P3. A Case Study of Defect Density and Change Density and their Progress over Time

Abstract for P3. See Section 5.2.

Results for P3. The results showed that the defect density of the reused framework was lower than the application. For the JEF framework, we found a *decreasing* defect density and a *decreasing* change request density. The DCF application showed a *decreasing* defect density, and an *increasing* change request density.

P5. A Case Study Comparing Defect Profile of a Reused Framework and of Applications Reusing the same Framework

Abstract for P5. The benefits of software reuse have been studied for many years. Several previous studies have observed that reused software has a lower defect density than newly built software. However, few studies have investigated empirically the reasons for this phenomenon. To date, we have only the common sense observation that as software is reused over time, the fixed defects will accumulate and will result in high-quality software. This paper reports on an industrial case study in StatoilHydro ASA, involving a reused Java class framework and two applications that use that framework. We analysed all trouble reports from the use of the framework and the applications, according to the Orthogonal Defect Classification (ODC), followed by a qualitative Root Cause Analysis (RCA).

Results for P5. Our results reveal that: 1) The framework has a much lower defect density in total than DCF and a slightly higher defect density than S&A. 2) The *higher* defect density of *function-type* defects of DCF and S&A is partially due to higher time-to-market pressure, more unstable requirements, and less quality control. 3) The *most severe* defects for JEF are *interface-type* and *assignment-type* defects. Since other applications such as DCF and S&A need to use the functions of the reused framework through its interface, interface-type defects of JEF may cause failure for several of the applications that reuse JEF. 4) Finally, our results showed us that impacts of defects on *capability* and *usability* are the most common in all three systems

Using the results of the study as a basis, we revised the explanatory model of the overall cause-effect relationship between software reuse and the lower defect density of reused software that was presented in Figure 5 into the model shown in Figure 9. Here, we have presented an improved overall cause-effect model between systematic reuse and lower defect density that will facilitate further studies and implementations of software reuse (see Figure 9). However, Figure 9 has been modified slightly in this thesis, compared to the original figure found in paper **P5**.

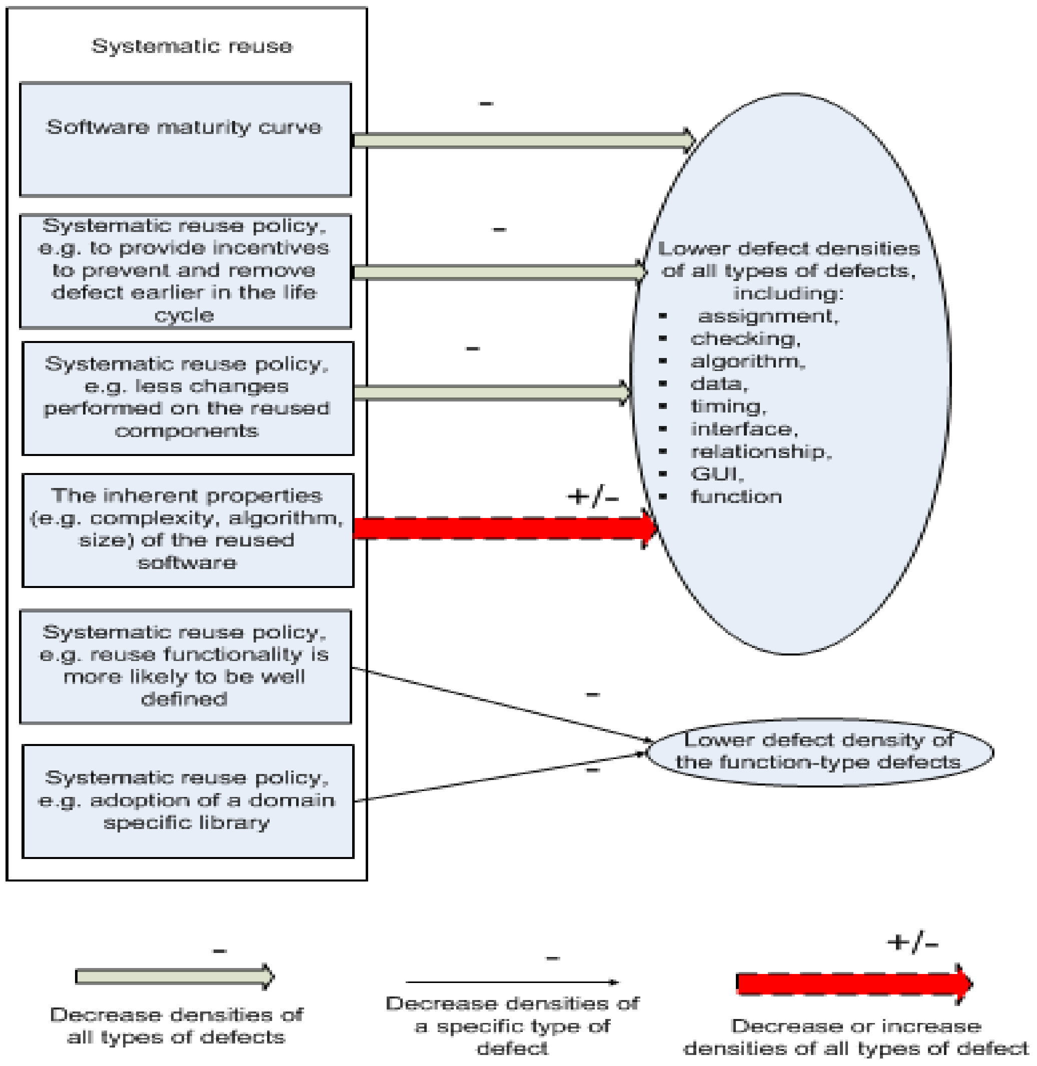
Systematic reuse
Software maturity curve
Systematic reuse policy, e.g. to provide incentives to prevent and remove defect earlier in the life cycle
Systematic reuse policy, e.g. less changes performed on the reused components
The inherent properties (e.g. complexity, algorithm, size) of the reused software
Systematic reuse policy, e.g. reuse functionality is more likely to be well defined
Systematic reuse policy, e.g. adoption of a domain specific library
-
+/-
Lower defect densities of all types of defects, including:
assignment,
checking,
algorithm,
data,
timing,
interface,
relationship,
GUI,
function
Lower defect density of the function-type defects
Decrease densities of all types of defects
Decrease densities of a specific type of defect
Decrease or increase densities of all types of defect

Figure 9. Improved overall cause-effect model between software reuse and the defect densities of reused software

Contributions of Study 3: The contribution of **P3** was to gain more knowledge about software evolution by investigating defect density, as well as how it evolves over time in relation to change request density for JEF and DCF. In paper **P5** the contribution was to increase our understanding of software reuse based on exploring defect profiles (i.e. defect density, density of specific defect types, and the severities/impacts of defects). This was related to the main contributions:

- **C2:** *Identification of differences/similarities of the defect profile for the reused framework vs. software reusing it,*
- **C3:** *Description of the software change lifecycle for the reused framework vs. software reusing it,* and
- **C4:** *Identification of possible software reuse improvements.*

5.4 Study 4: Analysing Change Data Related to the Source Code (Papers P4, P6)

Study 4 is similar to Study 3 when it comes to research design, but the qualitative RCA was performed for both **P4** and **P6**, where our observations have been presented. Our work in this study answers the research questions:

- **RQ1** (by **P4** and **P6**): *What is the relation between software changes and software reuse, by comparing the reused framework vs. software reusing it?*
- **RQ2** (by **P6**): *How do the reused framework and software reusing it evolve over time?*
- **RQ3** (by **P6**): *What improvements can be made towards the actual reuse practice at StatoilHydro ASA?*

P4. Experience Report on the Effect of Software Development Characteristics on Change Distribution
Abstract for P4. This paper reports on an industrial case study in StatoilHydro ASA involving a Java-class framework developed for reuse and an application that uses that framework. We analysed software changes from three releases of the framework and the application.

Results for P4. On the basis of our analysis of the data, we found that *perfective* and *corrective* changes account for the majority of changes in both the reused framework and the software reusing it, but it was only for JEF that adaptive changes followed closely. We have seen that DCF had a *poor structure* (i.e. complex and less maintainable code) in the beginning, and had to deal with frequent preventive changes before refactoring than after. *Designing for reuse* had an effect on the change types. Files related to *adaptive changes* stayed longer "open" before they were "closed", and were more frequently modified in JEF compared to DCF. Additionally, *preventive changes* were more common in DCF, due to the refactoring that took place (information gained from a qualitative RCA with the senior developer). Thus the amount of changes, as well as the effect on the localization of changes, was not similar to the systems that were not necessarily designed for reuse.

P6. Change Profiles of a Reused Class Framework vs. two of its Applications
Abstract for P6. Software reuse is expected to improve software productivity and quality. Although many empirical studies have investigated the benefits and challenges of software reuse from development viewpoints, few studies have explored reuse from the perspective of maintenance. This paper reports on a case study that compares software changes during the maintenance and evolution phases of a reused Java class framework with two applications that are reusing the framework.

Results for P6. The results reveal the following. 1) The reused framework was more stable, in terms of change density, than one application that was reusing it, and more unstable than the other. 2) The reused framework had profiles for change types that were similar to those of the applications, where *perfective changes* dominate. 3) The lifecycle of both the reused framework and its applications was the same: initial development, followed by a stage with extending capabilities and functionality to meet user needs, followed by a stage in which only minor defect repairs were made, and finally, phase-out. However, the reused framework went faster from the stage of extending capabilities to the stage in which only minor defect repairs were made than its applications. 4) The factors that affected the change densities and change profiles of both framework and applications were: *functionalities, development practice, complexity, size, and age*. Thus, all these factors must be considered to predict change profiles in the maintenance and evolution phase of software.

Contributions of Study 4: In paper **P4** the contribution was to study the impact that software changes had on different development characteristics (e.g. impact of reuse and impact of refactoring). This was related to the main contribution **C1:** *Identification of differences/similarities of the change profile for the reused framework vs. software reusing it.* In paper **P6** the contribution was to increase our understanding of maintenance benefits, and challenges of software reuse, based on exploring change profiles of the reused framework vs. software reusing it over time. This was related to the main contributions:

- **C1:** *Identification of differences/similarities of the change profile for the reused framework vs. software reusing it,*
- **C3:** *Description of the software change lifecycle for the reused framework vs. software reusing it,* and
- **C4:** *Identification of possible software reuse improvements.*

5.5 Overview of Contributions (C1- C4)

The identified contributions, as described in Section 1.6, relate closely to the studies we carried out in this thesis. We will briefly describe each contribution here, before moving on to the discussion part (Chapter 6) and relate the contribution with the overall research questions and state of the art.

C1: Identification of differences/similarities of the change profile for the reused framework vs. software reusing it (Papers P2, P4, P6)

We have identified the distributions of CRs, the data trend for how customers and developers assign priority to CRs, and maintainability of large components for the reused software. In addition, we also identified the impact that software changes had on different development characteristics (e.g. impact of reuse and impact of refactoring) for the reused framework (JEF) vs. software reusing it (DCF). Finally, we saw the differences/similarities in the maintenance activities for the reused framework (JEF) vs. software reusing it (DCF and S&A), by classifying the changes according to the definitions presented in Section 2.5, and we found that software reuse does not necessarily result in more stable software.

C2: Identification of differences/similarities of the defect profile for the reused framework vs. software reusing it (Paper P5)

We have identified differences/similarities in the defect profiles (in terms of defect density, density of specific defect types, and the severities/impacts of defects) for the reused framework (JEF) vs. software reusing it (DCF and S&A). Additionally, reused software may not have a lower defect density than non-reused software. Furthermore, software reuse will probably not reduce the density of the most severe defects either. The aspects of systematic software reuse that have helped to reduce the defect density of reused software were clearly defined requirements, solid design and testing, and cautions to changes.

C3: Description of the software change lifecycle for the reused framework vs. software reusing it (Papers P3, P6)

We have described how defect density and change request density and this relation progresses over time (i.e. over successive releases) for the reused framework (JEF) vs. software reusing it (DCF). We have also presented how the change profile for the reused framework (JEF) vs. software reusing it (DCF and S&A) evolves over time, according to the Bennett and Rajlich stage model for describing the lifecycle of a software system. Finally, our observation showed that both the reused framework and applications followed the so-called "80/20" rules.

C4: Identification of possible software reuse improvements (Papers P1, P5, P6)

Our findings resulted in identifying reuse benefits and factors that foster reuse in StatoilHydro ASA. We also presented an improved overall cause-effect model between systematic reuse and lower defect density that will facilitate further studies and implementations of software reuse. Finally, our observations explored factors of software that will affect the maintenance activities for the reused framework and software reusing it.

5.6 Overall Summary

Table 11 summarizes the main findings from Chapter 4 and Chapter 5, presented in relation to the four main studies.

Table 11. The studies and their relations to RQ, papers, contributions, research methods and validity comment

	Study 1	**Study 2**	**Study 3**	**Study 4**
Research Questions	RQ3	RQ1, RQ2	RQ1, RQ2, RQ3	RQ1, RQ2, RQ3
Papers	P1	P2, P3	P3, P5	P4, P6
Contributions	**C4:** Identified the reuse benefits and factors facilitating reuse.	**C1:** Presented the profile (i.e. distribution and maintainability) of CRs for the reused framework.	**C2:** Identified: 1) Reused software does not necessarily have lower defect density than software reusing it. 2) Software reuse will probably not reduce the density of the most severe defects.	**C1:** Identified: 1) Software reuse does not necessarily lead to stable software. 2) Perfective changes are the dominant change type for all three systems. 3) Designing for reuse has an effect on change types.
		C3: Observed that the CR density for JEF vs. DCF decreased after the first release.	**C3:** Observed: JEF: *decreasing* defect density and CR density. DCF: *decreasing* defect density, but an *increasing* CR density.	**C3:** Observed that JEF and DCF follow the same lifecycle, while S&A differs.
			C4: Presented a cause-effect model (Figure 9) between systematic reuse and lower defect density.	**C4:** Showed that development practices and functionality affect the maintenance activity the most.
Research Methods	Survey followed by semi-structured interviews.	Case study.	Case study.	Case study.
Validity Comment	Questions in the questionnaire have their origin from the research literature.	Categories to classify CRs and CR density are used in literature.	Defect density is described and used in literature.	Change density metric is described and used in literature.

6 *Evaluation and Discussion of results*

The chapter answers and discusses the three research questions **RQ1-RQ3** based on the results and the relations of contributions to the research questions and papers (Section 6.1). An overall discussion of the observed results is presented (Section 6.2). Further, the relations between the contributions to the state of the art (Section 6.3), the contributions to StatoilHydro generally (Section 6.4), and the contributions related to the SEVO goals (Section 6.5) are discussed. There is also a discussion of validity threats (Section 6.6) and the reflections on the research context (Section 6.7).

6.1 Research Questions Revisited: RQ1-RQ3

The answers and evaluation of the three research questions presented here are rather brief. For a more thorough discussion and evaluation, see the papers **P1**-**P6** in Appendix A.

RQ1: What is the relation between software changes and software reuse, by comparing the reused framework vs. software reusing it? (Contributions C1, C2)

Answering **RQ1** resulted in two major contributions, **C1** and **C2**, dealing with identifying the differences/similarities of the change profile (**C1**) and the defect profile (**C2**) of the reused framework vs. software reusing it.

- *The overall defect density of JEF (reused framework):* We analysed the TRs and performed a follow-up fish-bone Root Causal Analysis (RCA) by interviewing a senior developer who was familiar with development of both the JEF framework and the applications.
    - The results revealed the aspects of systematic software reuse that have helped to *reduce the defect density* of reused software in the company were: well-designed functionalities, solid design and testing, as well as cautions to changes (see also Section 6.2 for a more thorough explanation).

    - The relatively simple functionality and business logic of the reused software have also helped to *reduce the defect density* of the reused software.
    - The reused software had a large amount of GUIs that were not well implemented. These GUI-type defects partly lead to a *higher defect density* in total of the reused software than one of the applications reusing it, namely S&A.

- *Most frequent defect types for the reused framework and software reusing it:*
    - *Interface-type defects*[8] of JEF may cause failure for several of the applications that reuse JEF. This is because several other applications, e.g. DCF and S&A, need to use the functions of the reused framework through its interface.
    - The higher defect densities of *function-type defects*[9] in the DCF and S&A are due partially to higher time-to-market pressure, more unstable requirements, and less quality control.

- *The overall change density of JEF (reused framework):*
    - According to the fish-bone Root Causal Analysis (RCA) by interviewing the senior developer, the RCA showed us that developers in StatoilHydro ASA were cautious of involving changes into the reused framework, since changes could affect existing applications. This concern may therefore *reduce the change density* of the reused framework. Other possible explanations for the decreasing change density and CR density can be found in Section 6.2.
    - On the other hand, it is impossible to predict all future requirements of a framework developed for reuse. Unforeseen requirements of new applications may demand that many minor or major (such as refactoring) changes be made to the reused framework. This could explain the *higher change density* of the JEF compared to S&A.

- *The dominant change type for the reused framework and software reusing it:* We have seen that *perfective* changes were the dominant change type for both the reused framework and software reusing it, due to unclear and unstable requirements.

RQ2: How do the reused framework and software reusing it evolve over time? (Contribution C3)

Answering **RQ2** resulted in one major contribution, **C3**, dealing with describing the software change lifecycle for the reused framework vs. software reusing it.

[8] Communication problems between modules, components, device drivers, objects, functions via macros, call statements, control blocks and parameter lists [Chillarege92].

[9] The error should require a formal design change, because it affects significant capability, end-user interfaces, product interfaces, etc. The error occurred when implementing the state and capabilities of a real or an abstract entity [Chillarege92].

- *Evolution and maintenance of software developed for reuse:* Our observations revealed that factors that helped to reduce the number of changes that need to be made to the reused framework were good initial design and stable dependence on the part of software reusing it. The prime factor that may increase the number of changes that are made to the reused software is unpredictable contexts of usage.

- *Factors affecting maintenance:* Regarding change densities and change profiles of both framework and applications, the main factors from [Kemerer97] that affected the maintenance activity most in our case were *functionality* and *development practices*, followed closely by *complexity*. The factors that affected it least in our case were age and size.

- *The stage model:*
  - The change profiles of JEF and DCF were in line with the stage model (to describe the lifecycle of a software system) proposed by [Bennett00]: starting with initial development, followed by a stage with extending capabilities and functionality to meet user needs, followed by a stage in which only minor defect repairs are made, and finally, phase-out. However, the change profile of S&A was different. Our observations show that after the deployment of release 3 of S&A, two new users used the system and did acceptance tests. The results of the new acceptance tests led to many changes of all types. Hence, we see that the change profile was affected by the user profile. This indicates that for all software (developed *for* reuse and developed *with* reuse), developers need to prepare for all kind of changes when prospective new reusers come and evaluate the software.
  - We have observed that the reused framework went faster from the stage of extending capabilities to the stage in which only minor defect repairs were made than DCF. One explanation is that the JEF is a framework developed for reuse. After it has been used and reused by several applications, the company is cautious about making more changes to it.

RQ3: What improvements can be made towards the actual reuse practice at StatoilHydro ASA? (Contribution C4)

Answering **RQ3** resulted in one major contribution, **C4**, dealing with identifying possible software reuse improvements.

- *Ignorance of internal reuse training programmes:* Some of the developers in StatoilHydro ASA did not know about the existence of a reuse training programme (one of the results from our survey paper **P1**). The reason for this is that there was a training course for all developers who were involved in the reuse projects in the beginning, but some consultants joined the project after the course was held. Hence, these consultants did not get any training and were not aware of such programmes (this has also been discussed as a threat to the internal validity in Section 6.6). The study by Frakes and Fox [Frakes95] shows that corporate reuse training is rare. Even though the company has a reuse training programme, it still needs improvements. *So, StatoilHydro ASA must become better at promoting such training programmes, even after new consultants join the project.*

- *Need for experience sharing:* Developers revealed that a shared information and experience repository would be beneficial for software reuse (one of the findings from our survey paper **P1**). Thus, such a repository should be made available and ensured that it has relevant content and that it is used correctly. For instance, a wiki could be made for sharing information and experience.

- *Analysis of CRs/TRs for software process improvement:* StatoilHydro ASA should be more proactive to improve the current reuse process by analysing CRs and TRs. Our experience is that the recorded data is of poor quality, which makes any analysis hard, and also diminishes the usefulness of the CRs and TRs. Only through feedback from collected defect and change data can an organization "learn from its mistakes".

- *Need for improved, but still lightweight reporting schemes:* StatoilHydro ASA also needs to introduce an updated software change reporting scheme, so that more correct and more complete information is reported. In addition to the data in Rational ClearQuest (e.g. priority, severity, submission date of the change, testing type used) the company should report data such as:
    - defect type (e.g. based on a modified ODC),
    - change type (e.g. perfective, corrective, preventive and adaptive),
    - more precise effort data, e.g. according to an ordinal scale, such as:
        - small (minutes)
        - medium (hours)
        - large (days)
    - more fine-grained location information of software changes on the component level for applications reusing JEF, and
    - more fine-grained information on the actual development phase (e.g. requirements analysis, design, system test) when a software change is discovered.

6.2 Overall Discussion of the Observed Results

Based on our results from the three research questions, can we *indicate* that software developed *for* reuse is likely to be more *stable* vs. software developed *with* reuse, measured by a lower value of the three attributes defect density, CR density and change density? Table 12 gives an overview of how CR density, change density and defect density for JEF is compared to DCF and S&A from our studies.

Table 12. Overview of CR density, change density and defect density

CR density	**Change density**	**Defect density**
JEF < DCF	DCF >JEF > S&A	DCF >JEF > S&A

According to Table 12, JEF has *lower* CR density, change density and defect density compared to DCF, but *higher* compared to S&A. Therefore, we cannot indicate here if the reused framework is more stable than the software reusing it.

The following factors are derived from Figure 9, and can explain the lower defect density, CR density and change density for JEF over its three releases:

- *Software maturity curve:* Stable domain abstractions and proper packaging of functionalities are not discovered right away, but achieved through gradual re-design and possible re-engineering towards framework that may serve different applications. In our case JEF was *used* by PDM, but *reused* by DCF and S&A. Thus, the accumulated defect fixes will result in higher quality software [Lim94] over time. This also means that when JEF (or any other software) matures over time, there will be almost no changes in the user profiles and functionalities (i.e. software maturity). This will also result in decreasing defect density, CR density and change density for JEF.
- *Software reuse policy:* The lower defect density, CR density and change density of the reused framework is also partially due to the *systematic reuse policy* applied:
    - The reused framework had well-defined requirements, better design (e.g. product line or application families), and looser coupling with other software and lower complexities.
    - Reuse-oriented software will be thoroughly *tested* before it is approved for reuse [Baldassarre05]. Our results indicated that the reused framework went through several inspections to remove defects earlier in the life cycle.
    - Developers were cautious of involving *changes* into the reused framework, because the changes may affect existing applications. On the other side, it is difficult to predict all possible future requirements of a framework developed for reuse. That is, new requirements from software reusing the framework, which may ask for many minor or major (such as refactoring) changes of the reused framework.
- *The inherent properties (e.g. complexity, algorithm, size) of the reused software:* The inherent properties of the reused software may decrease or increase densities of all types of defects. For instance, our results indicated that the reused framework vs. software reusing it had *lower defect densities* for most of the defects because DCF and S&A are primarily business applications, with more rules and complex business logic. JEF had *higher defect density* of, e.g. *GUI-type defects*, simply because JEF has many more GUIs than the DCF and S&A, and so there are more requests to alter the layout of some of the JEF GUIs, especially concerning data displays, buttons, and checklists.

In addition to these aforementioned factors that are derived from Figure 9, we assume that the following factor can also affect the lower defect density, CR density and change density for JEF over its three releases:

- *Hierarchical system layers*: Reused software often lies between the operating system and other "utilities", and the application reusing it. The further down in the hierarchy the software lies, the fewer changes it will be exposed to (more stable software). For instance, the operating system and other software in the bottom will undergo minimal changes, as the software is well tested and kept stable. According to this reasoning, the software developed for reuse should undergo fewer changes compared to the applications, but more compared to the lower layers. We have not investigated this phenomenon by our data, but would assume that the number of changes and defects will be stabilized downwards in the hierarchy.

6.3 Discussion in Relation to the State of the Art

Our research on the relation between software changes of the reused framework vs. software reusing it has shown to be comparable with some previous literature, while contradicting others.

The results from our survey conducted in **P1** on the benefits of successful software reuse gave support to [Lim94] and [Frakes95]. An early observation on the distribution of CRs in **P2**, and even the defect density and CR density in **P3,** confirmed previous results such as [Mohagheghi04a] and [Mohagheghi04c]. Systematic software reuse has helped to reduce the defect density of reused software observed in **P5** through: well-designed functionalities, solid design and testing, as well as cautions to changes. This supports previous findings [Thomas97] [Frakes01] [Selby05]. Our data showed that the reused framework may not necessarily have a lower defect density than applications reusing the framework, and that the reused framework and applications reusing it have a similar density of the most severe defects, contradicting [Lim94] [Mohagheghi04c]. However, we observed in **P3** that the reused framework has decreasing defect density over its three releases, supporting [Ostrand05]. Even though Ostrand et al. [Ostrand05] do not explicitly mention the reuse impact as their central focus, they do show that defect density tends to decrease as the system matures.

Our findings in **P4** and **P6** reveal that *perfective* changes are the most common for both the reused framework and applications reusing the framework. Although slightly different definitions of change types have been used here, our results seem to support the observations of [Lientz78] [Jørgensen95] [Evanco99] [Satpathy02] [Mohagheghi04a] [Lee05]. However, other studies [Burch97] [Mockus00] [Schach03] gave different conclusions and showed that either the corrective or adaptive changes occur frequently.

We also observe that the reused framework has a higher change density than one application and a lower change density than the other in **P6**, which does not favour conclusions from any previous studies [Frakes01] [Algestam02] [Mohagheghi04a] [Selby05]. The change profile of JEF and DCF gave support to the simple/versioned stage model proposed by [Bennett00]. Both the reused framework and software reusing it support conclusions from [Schaefer85] [Kemerer97], i.e. about 80% of all work is caused by only 20% of all components.

6.4 Recommendations in general and specific to StatoilHydro ASA

In addition to the improvements towards the actual reuse practice mentioned in Section 6.1 for **RQ3**, the following recommendations should be considered by the company. Furthermore, the first two recommendations should be considered by the software practitioners in general:

- *Upgrade the defect and change reporting:* All software developing organizations have a basic software defect and change reporting system, but its use differs substantially. If the collected data are not being used by the organizations, or the data analyses are not done properly, as well as not given substantial feedback, this will lead to poor data quality for the analysis. This will be the case especially in those instances where the project members use a lot of time to report data correctly, and the data is never used in measurement programs. In our case, Rational ClearQuest and Rational ClearCase are only used to report and merge TR and CR. However, more systematic and extended use of the available data can easily be arranged for more thorough analysis (e.g. effort data).

- *Improved configuration management:* The way Rational ClearQuest and Rational ClearCase are configured should be improved. Our experience reveals that these two configuration management systems do not work perfectly together (Section 4.2.5). For instance, software changes reported in Rational ClearCase should be traced back (without any error messages) to the respective change request or trouble report submitted in Rational ClearQuest.

- *Improved O&S Masterplan (specific to StatoilHydro ASA):*
    - The O&S Masterplan is a project management template in StatoilHydro ASA, and the studied projects JEF, DCF and S&A have been developed according to this template. This document has been read by the managers in the company. Some of the main project's delivery parameters in this template are *quality, time and cost*. These parameters have not been clearly defined.
        - *Improvement:* To be able to say whether *quality* has improved, and *time and cost* have reduced or not, the company should define these parameters more precisely and how they are measured.
    - The main focus of the template does not come clearly. There is also a lack of motivation to drive the processes in the Masterplan forward.
        - *Improvement:* The Masterplan should be more precise on this point. The users should be more thoroughly informed about the content and motivation of such a template, e.g. through workshops and meetings..
    - The Masterplan has some incentives related specifically to development for reuse, but since it is a template for the managers' detailed guidelines they are not incorporated in the documentation.

        - *Improvement:* More specific recommendations towards reuse should be specified in the Masterplan.
    - The organizational regulations are strict, which again impacts the way business improvements are done.
        - *Improvement:* The organizational regulations should be less strict and more visible for the users in the company.

6.5 Relations to the SEVO Goals

The relations between the results and the SEVO goals, as defined in Section 1.2, are now considered:

G1. Better understanding of software evolution, focusing on CBSE technology. It is claimed that the work reported in this thesis advances the state of the art of software engineering as defined by its contributions. Better understanding of software maintenance and software reuse benefits (see Figure 9), as well as challenges of software reuse is achieved, as reflected in the contributions **C1**, **C2** and **C3**.

G2. Better methods to predict the risks, costs and profile of software evolution in CBSE. Although the goal **G2** has not been an explicit focus of this thesis, we have combined our results from **RQ1** and **RQ2** and the survey presented in **P1** to come up with improvements, e.g. in **C4** towards software reuse. Some feedback is also given to StatoilHydro ASA.

G3. Contributing to a national competence based around these themes, and G4. Disseminating and exchanging the knowledge gained. Most results are published, or planned on being published, and presented at international and national conferences and workshops. During this thesis work, several masters' students have directly or indirectly been involved in project work or masters' theses concerning software reuse and software evolution and maintenance in the SEVO project. Furthermore, the knowledge gained from our studies based on the data from StatoilHydro ASA has been disseminated back to them through written reports. This relates to all contributions **C1-C4**.

6.6 Brief Evaluation of Validity Threats

Some overall issues need to be discussed for the validity of the work in this thesis. Four groups of validity threats in empirical research are considered in Section 3.2.1 and validity comments for each of the four studies are discussed in Sections 4.3.1-4.3.4, as well as in each of the individual papers (see Appendix A). The collected data are gathered from the company's data repositories, and some threats to validity of our studies in this thesis and how these are handled are as follows:

Conclusion validity: Analysis in **P1** is based on an initial collection of data. Though it is too small to be statistically significant, it still yields interesting and valuable insights for us and for StatoilHydro ASA. The data set of change requests in **P2** should be sufficient to draw relevant and valid conclusions, but it is small. As new JEF releases are released, they should be included in our dataset to see if they support the same trend as discovered here.

Internal validity: Ambiguity could exist as to whether developers classify an incident as a Trouble Report or a Change Request, hence this remains a threat. In **P5** the interaction between JEF, DCF and S&A may lead to defects being attributed to the applications instead of to the framework, and hence defect reporting becomes a threat. Another threat to the internal validity is the number of files we have selected randomly from Rational ClearCase in **P4** and **P6**. The random sampling might have caused systematic bias. The randomness could have given us the files with the fewest changes, while the files with the most changes were left behind. Another possible threat to the internal validity is the ignorance of internal reuse training programmes (see Section 6.1 under **RQ3**). There were 16 developers at the time study 1 (paper **P1**) was conducted, and about 4 developers (out of these 16) did not get the necessary training. Even though the amount of developers who did not receive the reuse training is small, and all the documentation from the training course was made available, this still remains a threat to our findings in this thesis, since the effect of this is unknown.

Construct validity: One possible threat to construct validity in **P5** and **P6** is that we performed our causal analysis on a summary of all defects and changes. Given that we did not perform a detailed analysis of each defect and change, we may have missed important details. Another possible threat to this validity is that we asked only one senior developer during the RCA in **P4**, **P5** and **P6**, since we could not find more people who knew the details about all the three investigated systems.

External validity: The entire data set was taken from one company. The object of our studies for this thesis was a class framework, with only two applications. More similar studies need to be performed in different contexts and organizations in order to generalize our results. Our survey in **P1** is done completely by convenience sampling, and has a limited sample size. Thus, we cannot generalize outside of context.

6.7 Reflections on the Research Context

The SEVO research project is based on informal cooperation with the industrial partner, StatoilHydro ASA. When the SEVO project started it had no formal connections to any industrial partners, but we knew one of the project managers in the company. The project manager also has a position as an adjunct and associate professor at NTNU, and working with him previously had been a positive benefit. Therefore, we contacted him, and the feedback was positive. Hence, less effort was made in the beginning in order to initiate contact and agreement with organizations to collect research data, which many researchers struggle with.

All of our studies involved industrial data. Hence, our results were interesting not only to us, but also to the company that was involved. We were able to present our results to developers working in the company and received explanations on our observed data trends and results. In general, the company was involved in the paper writing process by providing feedback before submissions. A common issue through our industrial cooperation was that since we were external researchers who were just collecting and analysing existing data, we were not prioritized when times were busy. For instance, on one occasion we had to wait for 1.5 months to renew our access to the company, which

usually takes a couple of days. However, researchers often experience case studies taking another turn than anticipated, e.g. projects may be stopped or interrupted [Mohagheghi04b]. We have had a good collaboration with the company for four whole years in exploring data from the projects we were involved in, without the projects being stopped or interrupted. It has helped that StatoilHydro ASA is a very large and stable company. Therefore, we have had stable access throughout our research collaboration.
Our main research questions **RQ1**-**RQ3** (see Section 1.3) have been formulated based on collecting and combining existing data, mined from the company's repository. We have first collected the data and then formulated the research questions based on the collected data, a bottom-up approach. However, some of the fields in the recorded change requests and trouble reports (e.g. defect severity, effort estimation, subcomponent) were incomplete or missing. This indicates that the change requests and trouble reports have not been analysed properly or at all by the responsible people and that little feedback has previously been given on the collected change requests and trouble reports. If feedback were provided, the precision of data collection could be improved in the short run, and promising changes to the process could be suggested in the long run.

The problem with missing, incomplete and faulty data is not specific to this company, and the literature, e.g. [Mohagheghi06], reveals that most companies face similar challenges. That is, either no systematic metrics are defined or the metrics are not connected to quality goals, or there is a lack of resources and time to perform feedback or analyses. The overall feedback from conferences and workshops has been positive, and we have to admire StatoilHydro ASA's willingness to allow empirical studies of on-going projects.

7 Conclusions and Directions for Future Work

This thesis has presented the results of several empirical studies performed at StatoilHydro ASA, which is the largest oil and gas company in Norway. The studies have combined literature study, quantitative data from data repositories, and qualitative data from internal reports, semi-structured interviews, and qualitative RCA. The top-down confirmatory approach (with open-ended, research-defined metrics) was combined with the bottom-up explorative and descriptive approaches (with fixed and project defined metrics) [Mohagheghi03]. This work mainly analysed data that the company itself had not analysed at all, or not to the extent presented in this thesis.

The research carried out throughout this thesis has provided valuable insights into three main research questions, and resulted in four major contributions. Below we sum up our main conclusions (Sections 7.1-7.3), and outline possible future work (Section 7.4) based on our results.

7.1 Research Goal Revisited

Returning, finally, to our overall **RG** for this thesis:

- *Investigate the advantages/disadvantages of systematic software reuse and the reasons behind it, by analysing software change data. Then, based on these insights, propose specific reuse guidelines (as an example of improvements) to StatoilHydro ASA, as well as general recommendations to software practitioners.*

We found that applications reusing the framework usually face more unstable requirements, higher time-to-market pressure, and less quality control than the reused framework. Therefore, it is not surprising that they are more change-prone. However, reused software may not have a lower defect density and change density than non-reused software. Furthermore, software reuse will probably not reduce the density of the most severe defects either. So, developing a component for reuse will not automatically lead to better code quality. Our results revealed that, in order to lower the amount of software changes of the reused framework, it is important to define and implement a systematic reuse policy; clearly defined requirements, solid design and testing

[Succi01], and cautions to changes [Selby05]. Based on our results we have also proposed specific reuse guidelines (as an example of improvements) to StatoilHydro ASA (see Sections 6.1, 6.4 and 7.3.1), as well as general recommendations to software practitioners (see Sections 6.4, 7.2 and 7.3.2).

7.2 Possible Recommendations for Researchers on Software Reuse

We have presented an improved overall cause-effect model (see Figure 9) between systematic reuse and lower defect density that will facilitate further studies and implementations of software reuse.

- *Figure 9 indicates that a set of diverse factors have to be considered when discussing the relationship between software reuse and lower defect density* (paper **P5** in Appendix A, p.159).

Kemerer [Kemerer97] concludes that five main factors (i.e. software functionality, software complexity, development practices, software size, and software age) will affect maintenance activities (see paper **P6** in Appendix A). Regarding the change densities and change profiles of both the framework and the applications, those of Kemerer's factors [Kemerer97] that affect the maintenance activity most in our case are functionality and development practices, followed closely by complexity (see **RQ2** under section 6.1).

- *Researchers need to consider functionalities and development practices of the software, since they primarily influence the future change density and the type of change (perfective, corrective, adaptive, and preventive)* (paper **P6** in Appendix A, p.165).

7.2.1 From Internal Attributes to External Quality Properties

In the literature, defect density, CR density and change density have been used as measures for software quality [Mohagheghi04c], but these cannot be used as standard measures. We have chosen defect density, CR density and change density (*internal attributes*) as indicators of stability (*external quality property*) for the reused framework and software reusing it.

- *Although Li and Smidts [Li03] conclude that defect density and change request density correlate significantly with reliability (based on expert opinions), this still does not represent causality.*

7.3 Possible Recommendations to the IT industry on Software Reuse

7.3.1 StatoilHydro ASA Practitioners

Our findings show that there is much to gain by utilizing specific information about TRs, CRs and changes to the source code to support software reuse improvement. During our studies of TRs, CRs and changes to source code, we have collected, analysed and presented *direct metrics* (e.g. severity, priority) and *indirect metrics* (defect density, CR density and change density). We have also classified all the defects according to the modified ODC template (we have not used the whole template) and similarly for change requests and changes to the source code according to the four change types: perfective, corrective, preventive and adaptive (Section 2.5 for definition). However, we have seen that the recorded data is of mediocre quality, which makes it hard to have strong conclusions, and also diminishes the usefulness of the CRs and TRs.

- *Thus, it is important for the company to collect and monitor new data and associated measurements early (i.e. for the project at hand), so that they can improve their way of reporting software changes. We have in Section 6.1 (under* **RQ3***) suggested improvements towards the reuse practice in the company, as well as how more advanced use of the available data can easily be arranged for more thorough analysis (e.g. more precise effort data).*

Further, our findings in **RQ1** (section 6.1) revealed that the interface-type defects in the JEF may cause several of the applications that reuse the JEF framework to fail.

- *This indicates that more solid quality control or testing should be performed on reused software to reduce the possible interface defects* (paper **P5** in Appendix A, p.157).

It is important to estimate the change profile of a software system in order to arrange staff expertise, tools, and business strategies properly [Bennett00]. The results of our studies presented in **RQ2** (section 6.1) showed for the long-term evolution and maintenance of JEF:

- *Staff who understand the reused software well must be retained in the organization for a while after initial development*[10]*. Such action is necessary because the reused software may experience a stage in which its capabilities and functionality are extended to meet user needs, which will require making many major changes, after the initial deployment* (paper **P6** in Appendix A, p.183).

7.3.2 Software Practitioners in general

Our results in **RQ1** (section 6.1) showed that the lower defect density of JEF is partially due to the systematic implementation of the reuse policy, such as clearly defined functionality, solid design and testing [Succi01], and better management of changes (i.e. cautions to changes) [Selby05].

- *Thus, it is important for industrial practitioners to define and implement a systematic reuse policy to improve the defect density of reused software* (paper **P5** in Appendix A, p.157).

[10] We are aware that keeping the staff for a while in StatoilHydro ASA can be challenging, since most of the consultants are external. However, we believe that making the consultants stay will be beneficial for the company.

We have also uncovered incomplete fields in the recorded TRs and CRs. This indicates that the TRs and CRs have not been analysed properly by the responsible people and that little feedback has been given on the collected trouble reports and change requests.

- *If GQM feedback (e.g. to monitor and adjust new metrics and associated measurements) was provided from the start, the precision of the data collection could be improved in the short run and promising changes to the process could be suggested in the long run.*

Many software organizations are in possession of data resources concerning their own products and processes that they do not exploit fully. From the study conducted by [Morisio02] they report that very few companies have a reuse measurement program in place.

- *Through better recording of available information and simple analysis, many organizations could be able to focus on software reuse improvement initiatives better.*

7.4 Future Work

Software reuse and CBSE have advantages, but require a systematic approach in introducing each and in combining these. Possible directions for future work are:

7.4.1 Future Work Related to the State of the Art

- **Estimate software reliability by using defect density and change density:** To answer whether lower defect density and change density indicate more reliable and stable software over time, we should investigate whether defect-prone/change-prone components stay defect-prone/change-prone after deployment over several releases. Additionally, factors of software quality should also be measured over time (e.g. efficiency, maintainability, performance, usability). Further studies need to explore and estimate the above factors and others over time, for the reused framework vs. software reusing it from the same or other companies.

- **Longitudinal study of software evolution and maintenance:** Further studies should be performed, where software change data are collected from other applications reusing JEF. This could be used to validate our conclusions presented in this thesis.

- **Investigate the test processes for reused software vs. non-reused software:** One interesting question raised by our study is how to use different Quality Assurance (QA) methods to obtain a lower defect density of the reused software and the software that reuses it. Given that reused software has different defect types of the most frequent and severe defects from the software that reuses it, reused software may need to be tested in different ways than those used to test the applications that reuse it. This should be investigated further.

- **Study of effort distributions related to removing the causes of the most costly defects:** Further empirical studies of effort distributions related to those defects that are most costly to correct for reused vs. non-reused software should be performed. This could be used by the company to assign resources (e.g. number of persons, budget) in a more cost effective manner.

- **Investigate the amount of COTS or OSS used in software projects:** Separate studies indicate that 70% of the software companies use either COTS or OSS [Sommerseth06]. Further studies should be performed to explore how much of the projects in the studied company and other software companies use COTS or OSS.

7.4.2 Future Work Related to StatoilHydro ASA

- **Study of StatoilHydro ASA's reuse practice:** Further studies should be carried out to investigate how our recommendations are implemented in the company, and whether the recommendations have had the intended effect.

- **Investigate those aspects of software components that make them more or less fit for reuse:** Further studies should be performed to investigate the structure, configuration, interface, documentation and other aspects of components that make them more or less fit for reuse.

- **Measure the amount of changed source code:** Further studies should be performed to develop or adapt a scripting tool that can extract the amount of modified/deleted/added source code in Rational ClearCase for JEF, DCF, S&A and other applications reusing JEF. This was tried, but it did not lead to a permanent tool or method, since it had to be done manually.

- **Study of agile development methods vs. waterfall lifecycle models:** StatoilHydro ASA wants to explore whether the amount of source code decreases, by using agile methods instead of waterfall models in their projects.

- **Study of adopting COTS software:** StatoilHydro ASA is increasingly buying COTS software products, as are most companies. However, the COTS products that the company buys, have not always been 100% fitted to their needs. This has lead to tailoring of COTS products. Thus, like many other companies, when StatoilHydro ASA wants to upgrade the COTS software to the next release (typically 1-2 times per year), it faces a major merge/integration effort to incorporate the accumulated local software changes with the latest imported release. The company wants to identify how much effort and what amount of tailoring have been done to the COTS products, and what kind of "partnership" with the COTS producers might be needed.

8 Glossary

To address the relevant issues, we need reasonably precise definitions of the terms used. The following is a list of short definitions of some terms. Where relevant, they are re-iterated and elaborated in the thesis. These terms are mostly taken from [IEEE90].

Change Control Board (CCB): A group of people responsible for evaluating and approving or disapproving proposed changes to configuration items, and for ensuring implementation of approved changes [IEEE90].

Component: One of the parts that make up a *software* [emphasis added] system. A component may be hardware or software and may be subdivided into other components. *Note:* The terms "module," "component," and "unit" are often used interchangeably or defined to be sub-elements of one another in different ways, depending upon the context. The relationship of these terms is not yet standardized [IEEE90].

Component-Based Software Engineering (CBSE): The development of software by composing independent, deployable components [Sommerville04].

Configuration Management (CM): A discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements [IEEE90].

Error:

- That at least one (or more) internal state of the system deviates from the correct service state. The adjudged or hypothesized cause of an error is called a fault. In most cases, a fault first causes an error in the service state of a component that is a part of the internal state of the system, and the external state is not immediately affected. Many errors do not reach the system's external state and therefore cause failure [Avizienis04].
- The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result [IEEE90].

Failure (active, dynamic, external "error"):

- The non-performance or inability of the system or component to perform its intended function for a specified time under specified environmental conditions. A failure may be caused by design flaws – the intended, designed and constructed behaviour does not satisfy the system goal [Leveson95].
- The inability of a system or component to perform its required function within specified performance requirements [IEEE90].
- Since a service is a sequence of the system's external states, a service failure means that at least one (or more) external states of the system deviate from the correct service state [Avizienis04].

Fault (passive, static, internal "error"): An incorrect step, process, or data definition in a computer program. It is often called a bug or defect [IEEE90].

Java: An object-oriented programming language that was designed by Sun with the aim of platform independence [Sommerville04].

Maintainability: (1) The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. (2) The ease with which a hardware system or component can be retained in, or restored to, a state in which it can perform its required functions [IEEE90].

Metric: A quantitative measure of the degree to which a system, component, or process possesses a given attribute [IEEE90].

Quality:

- The degree to which a system, component or process meets customer or user needs or expectations [IEEE90].
- Ability of a set of inherent characteristics of a product, system or process to fulfil requirements of customers and other interested parties [ISO94].

Quality Assurance (QA): (1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which products are developed or manufactured [IEEE90].

Release: A version of a system or component to its customer or intended user [IEEE90].

Reliability:

- The characteristic of an item expressed by the probability that it will perform its required function in the specified manner over a given time period and under specified or assumed conditions. Reliability is not a guarantee of safety [Leveson95].
- Continuity for correct service [Avizienis04].

- The ability of a system or component to perform its required functions under stated conditions for a specified period of time [IEEE 90].
- A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time [ISO91].

SEVO: Software Evolution in Component-Based Software Engineering – a basic R&D project at NTNU in 2004-2008 under the ICT-2010 program at the Research Council of Norway, lead by Reidar Conradi. See http://www.idi.ntnu.no/grupper/su/sevo.html

Software: Computer programs, procedures and possibly associated documentation and data pertaining to the operation of a computer system [IEEE90].

Software engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1) [IEEE90].

Software evolution: The study of the ways an evolving software system changes [Sommerville04].

Software maintenance: (1) The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment (*adaptive maintenance; corrective maintenance; perfective maintenance*). (2) The process of retaining a hardware system or component in, or restoring it to, a state in which it can perform its required functions (*preventive maintenance*) [IEEE90].

Software Process Improvement (SPI):
- The process of making changes to a *software* [emphasis added] process with the aim of making the process more predictable or to improve the quality of its outputs. For example, if your aim is to reduce the number of defects in the delivered software, you might improve the process by adding new validation activities [Sommerville04].
- Software process improvement is a deliberate, planned methodology, following standardized documentation practices to capture on paper (and in practice) the activities, methods, practices, and transformations that people use to develop and maintain software and the associated products. As each activity, method, practice and transformation is documented, each is analysed against the standard of value added to the organization [Szymanski08].

Software reuse:
- The *systematic* [emphasis added] use of existing software, or software knowledge, to build new software [Wiki08].
- 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 [Morisio02].

System: An entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena. These other systems are the environment of the given system. The system boundary is the common frontier between the system and its environment [IEEE90].

9 References

[Abran91] Abran, A., Nguyenkim, H., Analysis of Maintenance Work Categories Through Measurement. In: Proceedings of the IEEE Conference on Software Maintenance. IEEE Computer Society Press, Sorrento, Italy, 1991, pp. 104-113.

[Adams84] Adams, E., Optimizing Preventive Service of Software Products. IBM Journal of Research and Development, 28(1): 2-14, 1984.

[Algestam02] Algestam, H., Offesson, M., Lundberg, L., Using components to increase maintainability in a large telecommunication system. In: Proceedings of the 9th Asia-Pacific Software Engineering Conference. IEEE Computer Society Press, Gold Coast, Australia, 2002, pp. 65-73.

[Avizienis04] Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C., Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing, 1(1): 11-33, 2004.

[Ayala07] Ayala, C., Sørensen, C.F., Conradi, R., Franch, X., Li, J., Open Source Collaboration for Fostering Off-The-Shelf Components Selection. In: Joe Feller and Alberto Sillitti (Eds.): Proceedings of the 3rd International Conference on Open Source Systems (OSS'07). Springer Verlag, Limerick, Ireland, 2007, pp. 17-30.

[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. Accessed 2008, http://www.sei.cmu.edu/

[Baldassarre05] Baldassarre, M.T., Bianchi, A., Caivano, D., Visaggio, G., An industrial case study on reuse oriented development. In: Proceedings of the 21st International Conference on Software Maintenance (ICSM'05). IEEE Computer Society Press, Budapest, Hungary, 2005, pp. 283-292.

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

[Basili87] Basili, V.R., Selby, R.W., Comparing the Effectiveness of Software Testing Strategies. IEEE Transactions of Software engineering, 13 (12): 1278-1296, 1987.

[Basili92] Basili, V., Caldiera, G., Rombach, H.D, Software Modeling and Measurement: The Goal Question Metric Paradigm. Computer Science Technical Report Series, CS-TR-2956 (UMIACS-TR-92-96), University of Maryland, College Park, MD, 1992, pp. 1-24.

[Basili93] Basili, V., Applying the Goal/Question/Metric Paradigm in the Experience Factory. In: Proceedings of the Tenth Annual CSR (Centre for Software Reliability) Workshop, Application of Software Metrics and Quality Assurance in Industry. Amsterdam, Holland, 1993, pp. 1-23.

[Basili94] 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.

[Basili96a] Basili, V., Briand, L., Condon, S., Kim, Y.M., Melo, W.L., Valett, J., Understanding and Predicting the Process of Software Maintenance Releases. In: Proceedings of the 18th International Conference on Software Engineering. IEEE Computer Society Press, Berlin, Germany, 1996, pp. 464-474.

[Basili96b] Basili, V.R., The role of experimentation in software engineering: past, current, and future. In: Proceedings of the 18th International Conference on Software Engineering (ICSE'96). IEEE Computer Society Press, Berlin, Germany, 1996, pp. 442-449.

[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. Accessed, 2008, http://www.sei.cmu.edu/

[Baumgartel59] Baumgartel, H., Using Employee Questionnaire Results for Improving Organizations: The Survey "Feedback" Experiment. In: Kansas Business Review, 1959, Vol. 12, pp.2-6.

[Belady76] Belady, L.A., Lehman, M.M., A model of a large program development. IBM Systems Journal, 15(1): 225-252, 1976.

[Bennet80] Bennet, P.L, Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations, Addison-Wesley, UK, 1980.

[Bennett00] Bennett, K.H., Rajlich, V.T., A Staged Model for the Software Life Cycle. IEEE Computer, 33(7): 66-71, 2000.

[Bertolino07] Bertolini, A., Software Testing Research: Achievements, Challenges, Dreams. In: Future of Software Engineering collocated with International Conference on Software Engineering (ICSE'07). IEEE Computer Society Press, Minneapolis, U.S., 2007, pp. 85-103.

[Boehm78] Boehm, B.W., Brown, J.R., Lipow, M., MacLeod, G.J., Merrit, M.J., Characteristics of Software Quality. Number 1 in TRW Series of Software Technology. Elsevier, North-Holland, 1978.

[Boehm06] Boehm, B., Value-Based Software Engineering: Overview and Agenda. In: S. Biffl (Eds.): Value-Based Software Engineering, Springer Verlag, Berlin, 2006, pp. 3-11.

[Brown98] Brown, A., Wallnau, W., Kurt. C. The Current State of CBSE. IEEE Software, 15(5): 37-46, 1998.

[Burch97] Burch, E., Kung, H.J., Modeling Software Maintenance Requests: A Case Study. In: Proceedings of the 13th International Conference on Software Maintenance (ICSM'97). IEEE Computer Society Press, Bari, Italy, 1997, pp. 40-47.

[Chapin01] Chapin, N., Hale, J.E., Khan, K., Ramil, J.F., Tan, W.G., Types of Software Evolution and Software Maintenance. Journal of Software Maintenance and Evolution: Research and Practice, 13 (1): 3-30, 2001.

[Chillarege92] Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.Y., Orthogonal Defect Classification - a Concept for in-Process Measurements. IEEE Transactions on Software Engineering, 18(1): 943-956, 1992.

[Conradi05] Conradi, R., Li, J., Slyngstad, O., Kampenes, V., Bunse, C., Morisio, M., Torchiano, M., Reflections on conducting an international survey of Software Engineering. In: Proceedings of the International Symposium on Empirical Software Engineering (ISESE'05). IEEE Computer Society Press, Noosa Heads (Brisbane), Australia, 2005, pp. 214-223.

[Cooper03] Cooper, D., Schindler, P., Business research methods, 8th edition, McGraw-Hill, 2003.

[Creswell94] Creswell, J.W., Research Design, Qualitative and Quantitative Approaches, Sage Publications, Beverly Hills, California, 1994.

[Creswell03] Creswell, J.W., Research Design, Qualitative, Quantitative, and Mixed Methods Approaches, Sage Publications, Beverly Hills, California, 2002.

[Crnkovic02] Crnkovic, I., Hnich, B., Jonsson, T., Kiziltan, Z., Specification, implementation, and deployment of components. Communications of the ACM, 45(10): 35-40, 2002.

[Dybå01] Dybå, T., Enabling Software Process Improvement: An Investigation of the Importance of Organizational Issues. PhD Thesis, NTNU 2001.

[Endres03] Endres, A., Rombach, D., A Handbook of Software and Systems Engineering, Empirical Observations, Laws, and Theories, Addison-Wesley, UK, 2003.

[Endres04] Endres, A., Rombach, D., A Handbook of Software and Systems Engineering; Empirical Observations, Laws and Theories, Addison-Wesley, UK, 2004.

[Evanco99] Evanco, W.M., Analyzing change effort in software during development. In: Proceedings of the 6th International Software Metrics Symposium (METRICS'99). IEEE Computer Society Press, Boca Raton (Florida), U.S., 1999, pp. 179-188.

[Fenton96] Fenton, N., Pfleeger, S.L., Software Metrics: A Rigorous & Practical Approach, 2nd edition, International Thomson Computer Press, 1996.

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

[Fenton00a] Fenton, N.E., Ohlsson, N.: Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Transaction on Software Engineering, 26(8): 797-814, 2000.

[Fenton00b] Fenton, N.E., Neil, M., Software Metrics: Roadmap. In: Anthony Finkelstein (Ed.): The Conference on the Future of Software Engineering. ACM Press, Limerick, Ireland, 2000, pp. 357-370.

[Finkelstein00] Finkelstein, A., Kramer, J., Software Engineering: a Road Map. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE'00). IEEE Computer Society Press, Limeric, Ireland, 2000, pp. 3-22.

[Fowler] Fowler, F.J., Survey Research Methods – Applied Social Research Series, Volume 1, Sage Publications, Beverly Hills, California, 1988.

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

[Frakes01] Frakes, W.B., Succi, G., An industrial study of reuse, quality, and productivity. Journal of Systems and Software, 57(2): 99-106, 2001.

[Godfrey00] Godfrey, M.W. and Qu, T., Evolution in Open Source Software: A Case Study. In: Proceedings of International Conference on Software Maintenance (ICSM'00). IEEE Computer Society Press, San Jose (California), U.S., 2000, pp. 131-142.

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

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

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

[IEEE90] IEEE Standard 610.12: Standard for Glossary of Software Engineering Terminology. IEEE Computer Society Press, 1990.

[IEEE93] IEEE Standard 1219: Standard for Software Maintenance. IEEE Computer Society Press, 1993.

[ISO91] ISO/IEC 9126-1: Standard for Information technology – Software product evaluation – Quality characteristics and guide-lines for their use, 1991.

[ISO94] ISO 9000-1: Quality management and quality assurance standards, Part 1: Guidelines for selection and use, 1994.

[IWPSE01] Tamai, T., Aoyama, M., Bennett, K., Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE'01), collocated with ESEC/FSE 2001. ACM Sigsoft, Vienna, Austria, 2001, p. 182.

[Jelinski72] Jelinski, Z, Moranda, P.B., Software Reliability Research. In: W. Freiberger (Ed.), Statistical Computer Performance Evaluation, Academic, New York, 1972, pp. 465-484.

[Johnson88] Johnson, R.E., Foote, B., Designing Reusable Classes. Journal of Object-Oriented Programming, 1(3): 26-49, 1988.

[Jørgensen95] Jørgensen, M., The quality of questionnaire based software maintenance studies. ACM SIGSOFT – Software Engineering Notes, 20 (1) 71-73.

[Karlsson95] Karlsson, E.A., Software Reuse A Holistic Approach, John Wiley & Sons, New York, 1995.

[Kemerer97] Kemerer, C.F, Slaughter S.A, Determinants of Software Maintenance Profiles: An Empirical Investigation. Journal of Software Maintenance, 9(4): 235-251, 1997.

[Kemerer99] Kemerer, C.F, Slaughter S.A, An Empirical Approach to Studying Software Evolution. IEEE Transactions on Software Engineering, 25(4): 493-509, 1999.

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

[Kitchenham99] Kitchenham, B.A., Travassos, G., Mayrhauser, A., Niessink, F., Schneidewind, N., Singer, J., Takada, S., Vehvilainen, R., Yang, H., Towards an Ontology of Software Maintenance. Journal of Software Maintenance: Research and Practice, 11(6): 365-389, 1999.

[Kraut96] Kraut, A.I., Organizational Surveys: Tools for Assessment and Change, Jossey-Bass, San Francisco, 1996.

[Krogstie06] Krogstie, J., Jahr, A., Sjøberg, D.I.K., A longitudinal study of development and maintenance in Norway: Report from the 2003 investigation. Information and Software Technology, 48(11): 993-1005, 2006.

[Kruchten01] Kruchten, P., The Nature of Software: What's so Special about Software Engineering?. The Rational Edge, 2001. Accessed 2008, http://www.therationaledge.com/

[Lee05] Lee, M.G., Jefferson, T.L., An Empirical Study of Software Maintenance of a Web-based Java Application. In: Proceedings of the 21st International Conference on Software Maintenance (ICSM'05). IEEE Computer Society Press, Budapest, Hungary, 2005, pp. 571-576.

[Lehman85] Lehman, M.M., Belady, L., Program Evolution: Processes of Software Change. Academic Press, London, 1985.

[Lehman96] Lehman, M.M., Laws of software evolution revisited. In: Proceedings of the European Workshop on Software Process Technology (EWSPT'96). Springer Verlag, Nancy, France, 1996, pp. 1-11.

[Lehman98a] Lehman, M.M., Perry, D.E., Ramil, J., On evidence supporting the FEAST hypothesis and the laws of software evolution. In: Proceedings of the 5th International Symposium on Software Metrics (METRICS'98). IEEE Computer Society Press, Bethesda, MD, 1998, p.84.

[Lehman98b] Lehman, M.M., Ramil, J.F., *F*eedback, *E*volution *A*nd *S*oftware *T*echnology – Some Results from the FEAST/1 Project. In: 11th International Conference on Software Engineering & its Applications, Preprints, Vol.1, Paris, France, 1998, pp.1-12.

[Lehman01a] Lehman, M.M., Ramil, J., Sandler, U., An approach to modelling long-term growth trends in software systems. In: Proceedings of the International Conference on Software Maintenance (ICSM'01). IEEE Computer Society Press, Florence, Italy, 2001, pp. 219-228.

[Lehman01b] Lehman, M.M., *F*eedback, *E*volution *A*nd *S*oftware *T*echnology – FEAST/2. Accessed 2008, http://www.doc.ic.ac.uk/~mml/feast2/

[Lehman01c] Lehman, M.M., Ramil, J., Rules and Tools for Software Evolution Planning and Management. Annals of Software Engineering, 11(1): 15-44, 2001.

[Lehman02] Lehman, M.M, Ramil, J.R., Software Evolution and Software Evolution Processes. Annals of Software Engineering, Vol. 14, pp. 275-309, 2002.

[Lehman05] Lehman, M.M., The role and Impact of Assumptions in Software Development, Maintenance and Evolution. In: 1st International Workshop on Software Evolvability collocated with International Conference on Software Maintenance (ICSM'05). IEEE CS Press, Budapest, Hungary, 2005, pp. 3-14.

[Leveson95] Leveson, N., Safeware: System safety and computers. Addison-Wesley, UK, 1995.

[Li03] Li, M., Smidts, C.S., A Ranking of Software Engineering Measures Based on Expert Opinion. IEEE Transactions on Software Engineering, 29(9): 811-824, 2003.

[Li04] Li, J., Conradi, R., Mohagheghi, P., Sæhle, O. A., Wang, Ø, Naalsund, E., Walseth, O. A., A Study of Developer Attitude to Component Reuse inside IT industries. In: F. Bomarius and H. Iida (Eds.): Proceedings of the 5th International Conference on Product Focused Software Process Improvement (PROFES'04). Springer Verlag, Kansai Science City, Japan, 2004, pp. 538-552.

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

[Lim94] Lim,W., Effect of Reuse on Quality, Productivity and Economics. IEEE Software, 11 (5): 23-30, 1994.

[Lyu96] Lyu, M. (Ed.), Handbook of Software Reliability Engineering, McGraw-Hill, New York, and IEEE Computer Society Press, Los Alamitos, 1996.

[McCall77] McCall, J.A., Richards, P.K., Walters, G.F., Factors in Software Quality. Technical Report RADC-TR-77-369, US Department of Commerce, 1977.

[McIlroy69] McIlroy, D., Mass-produced Software Components. In: Buxton, J.M., Naur, P., Randell, B. (Eds.): Proceedings of the Software Engineering Concepts and Techniques. 1968 NATO Conference on Software Engineering, 1969, pp. 138-155.

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

[Mockus00] Mockus, A., Votta, L.G., Identifying Reasons for Software Changes Using Historical Databases. In: Proceedings of the International Conference on Software Maintenance (ICSM'00). IEEE Computer Society Press, San Jose (California), U.S., 2000, pp. 120-130.

[Mohagheghi03] Mohagheghi, P., Conradi, C., Using Empirical studies to Assess Software Development Approaches and Measurement Programs. In: Proceedings of the 2nd workshop in Workshop Series on Empirical Software Engineering (WSESE'03), Rome, Italy, 2003, pp. 65-76. Issued by IESE in Kaiserslautern.

[Mohagheghi04a] Mohagheghi, P., Conradi, R., An Empirical Study of Software Change: Origin, Impact, and Functional vs. Non-Functional Requirements. In: Proceedings of the International Symposium on Empirical Software Engineering (ISESE'04). IEEE Computer Society Press, Redondo Beach (Los Angeles), U.S., 2004, pp. 7-16.

[Mohagheghi04b] Mohagheghi, P., The Impact of Software Reuse and Incremental Development on the Quality of Large Systems. PhD Thesis, NTNU 2004.

[Mohagheghi04c] Mohagheghi, P., Conradi, R., Killi, O.M., Schwarz, H., An Empirical Study of Software Reuse vs. Defect Density and Stability. In: Proceedings of the 26th International Conference on Software Engineering (ICSE'04). IEEE Computer Society, Edinburgh, Scotland, 2004, pp. 282-292.

[Mohagheghi04d] Mohagheghi, P., Conradi, R., Exploring Industrial Data Repositories. In: Coral Calero, Fernando Brito e Abreu, Geert Poels and Houari A. Sahraoui (Eds.): Proceedings of the 8th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'04). Springer Verlag, Oslo, Norway, 2004, pp. 61-77.

[Mohagheghi06] Mohagheghi, P., Conradi, C., Børretzen, J.A., Revisiting the Problem of Using Problem Reports for Quality Assessment. In: Kenneth Anderson (Ed.): Proceedings of the 4th Workshop on Software Quality collocated with International Conference on Software Engineering (ICSE'06). ACM Press, Shanghai, China, 2006, pp. 45-50.

[Mohagheghi07], Mohagheghi, P and Conradi, R., Quality, Productivity and Economic Benefits of Software Reuse: A Review of Industrial Studies. Journal of Empirical Software Engineering, 12(5): 471-516, 2007.

[Morisio02] Morisio, M., Ezran, M., Tully, C., Success and Failures in Software Reuse. IEEE Transactions on Software Engineering, 28(4): 340-357, 2002.

[Musa87] Musa, J.D., Iannino, A., Okumoto, K., Software Reliability: Measurement, Prediction, Applications, McGraw-Hill, New York, 1987.

[Musa96] Musa, J., Fuoco, G., Irving, N., Kropfl, D., Juhlin, B., The Operational Profile. In: M. Lyu (Ed.): Handbook of Software Reliability Engineering, McGraw-Hill, New York and IEEE Computer Society Press, Los Alamitos, 1996, pp. 167-216.

[Naur68] Naur, P., Randell, B., Software Engineering, Report on a Conference. NATO Scientific Affairs Division, Garmich, 1968.

[Neff66] Neff, F.W., Survey Research: A Tool for Problem Diagnosis and Improvement in Organizations. In: A.W. Gouldner and S.M. Miller (Eds.), Applied Sociology. New York: Free Press, pp. 23-38, 1966.

[Ostrand05] Ostrand, T.J., Weyuker, E.J., Bell, R.M., Predicting the Location and Number of Faults in Large Software Systems. IEEE Transactions on Software Engineering, 31(4): 340-355, 2005.

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

[Pfleeger94] Pfleeger, S., Experimental Design and Analysis in Software Engineering Part 1-5. ACM Sigsoft, Software Engineering Notes, 19(4): 16-20, 1994.

[Pigoski97] Pigoski, T.M., Practical Software Maintenance, John Wiley & Sons, New York, 1997.

[Pinsonneault93] Pinsonneault, A., Kraemer, K.L., Survey Research Methodology in Management Information Systems: An Assessment. Journal of Management Information Systems, 10(2): 75-105, 1993.

[Poore93] Poore, J., Mills, H., Mutchler, D., Planning and certifying software system reliability. IEEE Software, 10 (1): 88-99, 1993.

[Postema01] Postema, M., Miller, J., Dick, M., Including Practical Software Evolution in Software Engineering Education. In: Proceeding of the 14th Conference on Software Engineering. IEEE Computer Society, Press, Charlotte (North Carolina), U.S., 2001, pp. 127-135.

[Ramamoorthy82] Ramamoorthy, C.V., Bastani, F.B., Software Reliability: Status and Perspectives. IEEE Transactions on Software Engineering, 8(4): 354-371, 1982.

[Robson93] Robson, C., Real World Research: A Resource for Social Scientists and Practitioners-Researchers, Blackwell, Oxford, 1993.

[Rothenberger03] Rothenberger, M.A., Dooley, K.J., Kulkarni, U.R., Nada, N., Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices. IEEE Transactions on Software Engineering, 29(9): 825-837, 2003.

[Satpathy02] Satpathy, M., Siebel, N.T., Rodríguez, D., Maintenance of Object Oriented Systems through Re-engineering: A Case Study. In: Proceedings of the 10th International Conference on Software Maintenance (ICSM'02). IEEE Computer Society Press, Montreal, Canada, 2002, pp. 540-549.

[Schach03] Schach, S.R., Jin, B., Yu, L., Heller, G.Z., Offutt, J., Determining the Distribution of Maintenance Categories: Survey versus Management. Journal of Empirical Software Engineering, 8 (4): 351-366, 2003.

[Schaefer85] Schaefer, H., Metrics for optimal maintenance management. In: Proceedings Conference on Software Maintenance. IEEE Computer Society Press, Los Alamitos, CA, 1985, pp. 114-119.

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

[Selby05] Selby, W., Enabling Reuse-Based Software Development of Large-Scale Systems. IEEE Transactions on Software Engineering, 31(6): 495-510, 2005.

[Selnes99] Selnes, F., Markedsundersøkelser, 4 utgave, Tano Aschehoug, 1999.

[SEVO04] The Software EVOlution (SEVO) Project, 2004-2008. Accessed 2008, http://www.idi.ntnu.no/grupper/su/sevo/

[Sindre95] Sindre, G., Conradi, R., Karlsson, E., The REBOOT Approach to Software Reuse. Journal of System Software*,* 30 (3): 201–212, 1995.

[Sommerseth06] Sommerseth, M., Component based system development in the Norwegian software industry. NTNU master thesis. Accessed 2008, http://www.idi.ntnu.no/grupper/su/su-diploma-2006/sommerseth-dipl06.pdf

[Sommerville98] Sommerville, I., Dependability, 1998. Accessed 2008, http://www.comp.lancs.ac.uk/computing/resources/IanS/Ian/Courses/CritSys-2004/PDF-notes/Dependability-notes.pdf

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

[Sommerville04] Sommerville, I., Software Engineering, 7th edition, Addison-Wesley, UK, 2004.

[Sousa98] Sousa, M., Moreira, H., A Survey on the Software Maintenance Process. In: Proceedings of the 14th IEEE International Conference on Software Maintenance. IEEE Computer Society Press, Bethesda (Maryland), U.S., 1998, pp. 268-274.

[Succi01] Succi, G., Benedicenti, L., Vernazza, T., Analysis of the Effects of Software Reuse on Customer Satisfaction in an RPG Environment. IEEE Transactions on Software Engineering, 27(5): 473-479, 2001.

[Swanson76] Swanson, E.B., The Dimensions of Maintenance. In: Proceedings of the 2nd IEEE International Conference on Software Engineering. IEEE Computer Society Press, San Francisco (California), U.S., 1976, pp. 492-497.

[Szymanski08] Szymanski, D.J., Neff, T.D., Defining Software Process Improvement. Accessed 2008, http://www.stsc.hill.af.mil/crosstalk/1996/02/defining.asp

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

[Thomas97] Thomas, W.M., Delis, A., Basili, V.R., An analysis of errors in a reuse-oriented development environment. Journal of Systems and Software, 38(3): 211-224, 1997.

[Trochim08] Trochim, W.M.K, "Research Methods Knowledge Base". Accessed 2008, http://www.socialresearchmethods.net/kb/sampterm.htm

[Vliet01] Vliet, H., Software Engineering: Principles and Practice, 2nd edition, John Wiley & Sons, New York, 2001.

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

[Wiki08] Wikipedia on Software Reuse. Accessed 2008, http://en.wikipedia.org/wiki/Software_reuse

[Wohlin00] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering – An Introduction, Kluwer Academic Publishers, 2000.

[Yin94] Yin, R.K., Case Study Research Design and Methods, Sage Publications, Beverly Hills, California, 1994.

[Yin03] Yin, R.K., Case Study Research, Design and Methods. Sage Publications, Beverly Hills, California, 2003.

[Yip94] Yip, S., Lam, T., A Software Maintenance Survey. In: Proceedings of the 1st IEEE Asia-Pacific Software Engineering Conference. IEEE Computer Society Press, Tokyo, Japan, 1994, pp. 70-79.

[Zelkowitz98] Zelkowitz, M.V., Wallace, D.R., Experimental Models for Validating Technology. IEEE Computer, 31(5): 23-31, 1998.

Appendix A: Selected papers

In this Appendix we have included the six papers that have contributed towards the work presented in this thesis. We present them here in chronological order. The papers are:

- **P1:** *An Empirical Study of Developers' Views on Software Reuse in Statoil ASA*
- **P2:** *An Empirical Study of Software Changes in Statoil ASA – Origin, Priority Level and Relation to Component Size*
- **P3:** *A Case Study of Defect-Density and Change-Density and their Progress over Time*
- **P4:** *Experience Report on the Effect of Software Development Characteristics on Change Distribution*
- **P5:** *A Case Study Comparing Defect Profiles of a Reused Framework and of Applications Reusing it*
- **P6:** *Change Profiles of a Reused Class Framework vs. two of its Applications*

Except for minor formatting adjustments, the papers are presented in their original version in this thesis.

P4: Experience Report on the Effect of Software Development Characteristics on Change Distribution

Published: In Andreas Jedlitschka and Outi Salo (Eds.): Proc. 9th International Conf. on Product Focused Software Development and Process Improvement (PROFES'08), 23-25 June 2008, Rome, Italy. Springer Verlag, ISBN 978-3-540-69564-6, pp. 158-173. The paper received the Best Paper Award at the conference, and was invited and later accepted for a special issue of Software Process Improvement and Practice (SPIP) journal.

Anita Gupta[1], Reidar Conradi[1], Forrest Shull[2], Daniela Cruzes[2], Christopher Ackermann[2], Harald Rønneberg[3] and Einar Landre[3]

[1] Dep. of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU), Trondheim, Norway
{anitaash, conradi}@idi.ntnu.no
[2] Fraunhofer Center Maryland, College Park, USA
{fshull, dcruzes, cackermann}@fc-md.umd.edu
[3] StatoilHydro ASA KTJ/IT, Forus, Stavanger
{haro, einla}@statoilhydro.com

Abstract. This paper reports on an industrial case study in a large Norwegian Oil and Gas company (StatoilHydro ASA) involving a reusable Java-class framework and an application that uses that framework. We analyzed software changes from three releases of the framework and the application. On the basis of our analysis of the data, we found that perfective and corrective changes account for the majority of changes in both the reusable framework and the non-reusable application. Although adaptive changes are more frequent and has longer active time in the reusable framework, it went through less refactoring compared to the non-reusable application. For the non-reusable application we saw preventive changes as more frequent and with longer active time. We also found that designing for reuse seems to lead to fewer changes, as well as we saw a positive effect on doing refactoring.

Keywords: Software reuse, Software quality, Software changes, Case Study

1 Introduction

Understanding the issues within software evolution and maintenance has been a focus since the 70's. The aim has been to identify the origin of a change, as well as the frequency and cost in terms of effort. Software changes are important because they account for a major part of the costs of the software. At the same time, they are necessary; the ability to alter software quickly and reliably means that new business opportunities can be taken advantage of, and that businesses thereby can remain competitive [1].

Several previous studies have concluded that reusable software components are more stable (less change density) than non-reusable components [20-22]. However, few of these studies have investigated and compared the characteristics of software changes (such as distribution, how long the changes tend to stay in the system, and number of files modified for each change type) for reusable and non-reusable components. In the study described here we investigate whether aspects of software changes, such as their frequency, type, or difficulty, can be better understood based on:

- Characteristics of the process being applied (e.g. whether different change characteristics are seen when designing for reuse vs. designing for a specific context), and
- Characteristics of the product being built (e.g. whether different change characteristics are seen for systems before and after refactoring).

By "change characteristics" here we refer to attributes of the set of software changes made to a system over time, such as the relative frequency of different types of changes, the files of the system affected by the changes, and how the changes were implemented.

The case study described here is on the reuse process in the IT-department of a large Norwegian Oil & Gas company, StatoilHydro ASA[1]. We have collected data from software changes for three releases of a reusable class framework called Java Enterprise Framework (JEF), as well as three releases of one application called Digital Cargo Files (DCF) that uses this framework "as-is", without modification. All data in our study are software changes from the evolution (e.g. development) and maintenance phases for the three releases each of two systems.

The purpose of this study is to compare change characteristics across systems, with respect to the impact of reuse on change types, frequency, active time and localization of the effects of changes on the systems.

We were particularly interested in gaining insight into properties of systems being designed to be reusable, since that was a major focus for the reuse program at StatoilHydro ASA. The results are important in that they characterize and explain the changes to the reusable framework and the non-reusable application.

The paper is structured as follows. Section 2 presents the related work. Section 3 introduces the context in StatoilHydro ASA. Section 4 presents the motivation for the research and the research questions. Furthermore, Section 5 describes the research methodology. Section 6 presents the results and possible explanations of our analysis of software changes extracted from Rational ClearCase. Section 7 looks into the validity threats for our study. Section 8 states our conclusions.

[1] ASA stands for "allmennaksjeselskap", meaning Incorporated.

2 Related work

Lehman [2] carried out the first empirical work on software changes, finding that systems that operate in the real world have to be adapted continuously, otherwise, their changeability decreases rapidly. During the lifetime of software systems, they usually need to be changed as the original requirements may change to reflect changing business, user and customer needs [3]. Other changes occurring in a software system's environment may emerge from undiscovered errors during system validation that require repair, from the introduction of new hardware.

Changes to software may be categorized into four classes based on the intent of making the change, namely corrective, adaptive, perfective and preventive. In general, corrective refers to fixing bugs, adaptive are related to new environments or platforms, while implementing altered or new requirements, as well as improving performance, can be classified as perfective. Finally, changes made to improve future maintainability can be thought of as preventive [4]. Such taxonomy is useful because it captures the kind of situations that developers face over time. However, differences may exist in the definition of these change classes, which can make the comparison of studies difficult. We have in our study decided to use the definition given by [5]:

- *Adaptive* changes are those related to adapting to new platforms, environments or other applications.
- *Corrective* changes are those related to fixing bugs.
- *Perfective* changes are that that encompass new or changed requirements as well as optimizations.
- *Preventive* changes are those having to do with restructuring and reengineering.

Several studies have investigated the distributions of different changes (e.g. corrective, adaptive, perfective, and preventive) based on change logs of different systems. These studies show that:

- *The classifications of changes are different among different studies.* For example, some studies [6-11] have classified the changes into adaptive, corrective, and perfective; some of them have still a fourth category [9-11]. Other studies have classified changes into adaptive, preventive, and perfective [12-17] and four of these studies have classified changes into a fourth category: corrective [14-17]. One study has classified changes into planned enhancement, requirement modifications, optimization and "other" [18]. Yet another study has classified changes into user support, repair and enhancement [19].
- *Definitions of change types are different among different studies.* For example, perfective change has been defined to include user enhancements, improved documentation, and recoding for computational efficiency [6][7]. It is also defined as encompassing new or changed requirements (expanded system requirements) as well as optimization [12][13][15]. While, [10] has defined the same type as including enhancements, tuning and reengineering.
- *The distributions of changes are different for different systems.* For example, the most frequent changes of the studied system in [6][10][11] are perfective changes. However, perfective changes in the system in [7] are the least frequent ones. In the study conducted by [9][15] the most frequent changes are adaptive changes. While, in [18] the most frequent changes are in the category "other".

Table 1 shows different studies and the most frequent changes found in the results. We also distinguish systems that were designed to be reused as part of another system. We can see that 64% of the studies have perfective changes as the most frequent ones, 21% have corrective changes, followed closely by 14% that have adaptive changes as the most frequent ones.

Other studies [20-25] have investigated whether the amount of changes varies according to development characteristics without classifying changes into different categories. We are aware of no previous studies that have compared change distributions between reusable software components and non-reusable ones, which we are looking at in this study.

Table 1. Related work

Reusable system?	**Studied systems**	**Most frequent change types**
No	System which has been operational for at least 1 year, represents a significant investment of time and effort, and is of fundamental importance to the organization [6].	Perfective changes
No	A case study investigating 2152 change requests collected for 2 years in a Canadian financial institute [9].	Adaptive changes
No	A survey conducted in the MIS department in 9 different business types in Hong Kong [10].	Perfective changes
No	Survey carried out in a computer department of a large Norwegian organisation in 1990-1991 (study1) and 1992-1993 (study2). The computer department studied maintains of more than 70 software applications and include 110 maintainers, distributed on 11 maintenance groups [14].	Perfective changes
No	Study of 10 projects conducted in Flight Dynamic Division (FDD) in NASA's Goddard Space Flight Center. FDD maintains over 100 software systems totaling about 4.5 millions lines of code [11].	Perfective changes
No	Analyzed 654 change and maintenance requests from a large software application (written in SQL) [19]	Corrective changes
No	A survey carried out in financial organizations in Portugal. Data was collected from 20 project managers [15].	Adaptive changes
No	453 non-defect changes from an Ada system developed at the Software Engineering Laboratory (SEL) of the NASA Space Flight Center [18].	Perfective changes
No	Version control and maintenance records from a multi-million line real-time software system [7].	Corrective changes
No	An integrated system for automated surveillance, a reengineering project (Written in C++; version 3 is 41 KLOC) [16].	Perfective changes
No	Three software products, a subset of Linux consisting of 17 kernel modules and 6506 versions, and GCC consisting of 850 KLOC [8].	Corrective changes
Yes	Analyzed 169 change requests (covers any change in the requirements or assets from the time of requirement baseline) for 4 releases of a large telecom system. This system reuses components [12].	Perfective changes
No	Web-based java application, consisting of 239 classes and 127 JSP files. Analysis of fault reports [17].	Perfective changes
Yes	Analyzed 208 change requests (covers any change in the requirements) for three releases of a reusable framework [13].	Perfective changes

3 The StatoilHydro ASA setting

StatoilHydro ASA is a Norwegian company, and is part of the Oil & Gas industry. It is represented in 40 countries, has a total of about 31,000 employees, and is headquartered in Europe.

The central IT-department of the company is responsible for developing and delivering software meant to give key business areas better flexibility in their operation. It is also responsible for the operation and support of IT-systems. This department consists of approximately 100 developers, located mainly in Norway. Since 2003, a central IT strategy of the O&S (Oil Sales, Trading and Supply) business area has been to explore the potential benefits of reusing software systematically. StatoilHydro ASA has developed a custom framework of reusable components based on J2EE - JEF (Java Enterprise Framework). The actual JEF framework consists of seven separate components, which can be applied separately or together when developing applications. Table 2 shows the size and release date of the three JEF releases. This JEF framework is currently being reused in two applications at StatoilHydro ASA. In this study we investigated one of these applications, namely DCF (Digital Cargo Files), due to the available data set. DCF is mainly a document storage application: A “cargo file” is a container for working documents related to a deal or cargo, used by all parties in the O&S strategy plan at StatoilHydro ASA. DCF is meant to replace the current means of handling cargo files, which are physical folders containing printouts of documents pertaining to a particular cargo or deal. The DCF application also consists of seven components. Table 3 gives an overview of the size and release date of the three DCF releases.

Although they have different aims, JEF and DCF have certain similarities**.** These systems operate in the same business domain, were conducted by a fairly stable set of developer from the same IT-department, were built over nearly the same time period, and are of similar size. The maturity level is the same for JEF and DCF. Thus they provide us with a fairly controlled environment for looking at whether process and product considerations impact the change characterization of systems.

Table 2. The size and release date of the three JEF releases

Release 1: 14. June 2005	Release 2: 9. Sept. 2005	Release 3: 8. Nov. 2005
17 KSLOC	19 KSLOC	20 KSLOC

Table 3. The size and release date of the three DCF releases

Release 1: 1. Aug. 2005	Release 2: 14. Nov. 2005	Release 3: 8. May 2006
20 KSLOC	21 KSLOC	25 KSLOC

3.1 Software change data in StatoilHydro ASA

When a software change is detected during integration/system testing, a change request or trouble report is written (by test manager, developers etc.) and tracked in the Rational ClearQuest tool. Examples of software changes are:

- add, modify or delete functionalities

- address a possible future problem by improving the design
- adapt to changes from component interfaces
- bug fixing

The project leader or test managers assign the change requests and trouble reports to developers. The developers then access the source files in the version control system (Rational ClearCase) to make modifications. When implementing the changes the developers adhere to the following steps:

(1) Check out the file(s) corresponding to the specific change request.
(2) Implement the specific software change.
(3) Check the file back in to Rational ClearCase.
(4) While checking in the file, they input a change description, a thorough description of what changes were made and a time and date.

Rational ClearCase captures various information about source code changes and the ClearQuest also stores information about changes to requirements and other documents. We extracted the data for JEF and DCF from Rational ClearCase as described in Table 4, with a corresponding example.

Table 4. The data collected from Rational ClearCase

Data	Example
File id	8
System	JEF
Filename	DataAccessException
Number of versions	2
Dates[2]	Version 1: 19.04.2005, Version 2: 04.01.2007
Physical size (kilobytes)	1800
Size of a files **first version**	Non-commented SLOC (source lines of code): 34 Commented SLOC: 58
Size of a files **last version**	Non-commented SLOC: 34 Commented SLOC: 51
Descriptions of what changes occurred in each file version	Version 1: Component support for accessing data. Version 2: Remove obsolete java source file header entries.
Component to which the file belongs	One of the seven JEF or DCF components

4 Research questions

The existence of comparable systems in the StatoilHydro ASA environment gave us the ability to examine our major research goal: *The impact of reuse:*

- The reusable framework (JEF) had changes related to all kinds of potential downstream reuse.
- The non-reusable application: DCF had only software changes related to the specific goals of that application (explained in section 3). The DCF application has different development characteristics for release 1 and release 2 and 3:

[2] The date here refers to when the file was checked in after undergoing a change by the developer.

- DCF1.0 is relatively unstructured, since it was unclear what the developers were supposed to implement, and how it should be organized. In the beginning the developers did not have a detailed design, and a lot of changes were made regarding functionality and design during the implementation and testing period.
- DCF 2.0 and 3.0 were based on refactoring. Prior to DCF2.0, when the design and the goals became clearer the developers realized that the code they had developed was complex and hard to maintain. Therefore, they decided to do refactoring to improve the structure and ease the code maintenance.

The research questions we addressed for our goal are:

RQ1: Does the distribution of change types vary for different development characteristics? We hypothesize that the development process being employed would have a measurable impact on the type and number of changes required to a system. Making a software reusable may help to reduce certain kinds of changes, but may increase the frequency of other kinds of changes. For example, components that need to be reusable may have more adaptive changes, over a longer period of time, as more environments are found that could exploit such components. Since DCF went through a refactoring we also expect the preventive changes to decrease for release 2 and 3, compared to release 1. We have the following related questions:

- *RQ1.1: Does JEF have higher adaptive changes than DCF?*
- *RQ1.2: Is there a decrease in the preventive changes before and after refactoring for DCF?*
- *RQ.1.3 Do perfective and corrective changes account for the majority of the changes, with adaptive following closely?*

RQ2: What change types are the longest active for different development characteristics? Our purpose is to investigate what change types (perfective, preventive, corrective and adaptive) are longest active for different systems, which may provide some insight into which types of changes are more difficult or problematic to make. It is important to clarify that the changes that are longest active do not necessarily require more effort; a change may not have been constantly under work the entire time it was open. However, if characteristic patterns are found, this can be useful as the starting point for a conversation with the developers to explore differences. The following are the related research questions for RQ2:

- *RQ2.1: Are adaptive changes longer active in JEF than DCF?*
- *RQ2.2: Are preventive changes longer active before refactoring than after for DCF?*

RQ3: How localized are the effects of different types of changes, for different development characteristics? We hypothesize that a change that needs to modify many files is not well-supported by the architecture, and hence more expensive to fix. Our purpose is to investigate whether development changes can be successful in reducing this metric for a system, and allowing future changes to be more localized. We would like to investigate the following research questions for RQ3:

- *RQ3.1: Is the average number of files modified for adaptive changes higher in JEF than DCF?*

- *RQ3.2: Is the average number of files modified for preventive changes higher before refactoring than after for DCF?*

5 Research methodology

We analyzed a representative sample of the software changes for both the JEF framework and the DCF application to answer the research questions *RQ1-RQ3*.

Our analysis began from the files checked into Rational ClearCase. In total over all releases, there were 481 files for JEF framework and 1365 for the DCF application, distributed across the seven components in each system. Due to the manual nature of the analysis it was infeasible to analyse all changes to all 1846 files. Therefore we adopted a strategy of analysing a representative subset of the files in each component. In our data collection we decided to have a threshold value of 10 files. This means that if a component had more than 10 files we would not include all of the files in our dataset, but pick a random subset that was representative of the properties of the largest. A sampling calculator [26] was used to calculate a sufficient sample size. For example component JEFClient had 195 files. Based on the calculated sample size (165), we randomly (using a mathematic function in excel) selected 165 files from the JEFClient to include in the dataset.

In total we used 442 files for the JEF framework and 932 files for the DCF application. Table 5 gives an overview of the actual number of files in Rational ClearCase vs. the number of files we analyzed, and the size (in SLOC, including the non-commented source lines of code) for the collected files.[3] In total we analyzed 1105 changes for the JEF framework and 4650 changes for the DCF application. We can see that the number of changes for DCF is higher than for JEF. This can be explained by that DCF development was going on for about 10 months (Table 3), while JEF development was going on for about 6 months (Table 2). Due to longer development period, DCF faced more changes.

Table 5. Description of data set collected from ClearCase

	Actual number of files	**Number of files collected**	**Number of changes collected**	**Size in SLOC for files collected**
DCF: Release 1 (before refactoring)	426	282	2111	15K
DCF: Release 2 and 3 (after refactoring)	939	650	2539	55K
JEF framework	481	442	1105	38K
Total	**1846**	**1374**	**5755**	**108K**

During the classification and comparison, we noticed that some of the changes descriptions were labelled as "no changes" (meaning no changes were made to the code), and "initial review" (meaning changes resulting from formal code review of

[3] However, the SLOC is just for the last version of the collected files. For example, if a file has 6 versions, the SLOC is presented for version 6 only and not for the remaining files. Thus these values should be taken as only an approximate overview of file sizes.

the code). The changes in category “code review” are changes we cannot classify, since no description of the change was provided. We grouped “no changes” into the category “other” and “initial review” into the category “code review”. The changes in the category “other” and “code review” are excluded from the analysis for RQ1 – RQ3. Quantitative differences among the change profiles of the systems were used to formulate questions for the development team. These questions were addressed via interviews which elicited possible explanations for these differences.

6 Results

Before investigating our specific research questions, we examined the distribution of data across the change history. The test for normality of our datasets failed, meaning that the data is not normally distributed. Additionally, we investigated the variances for each change type for JEF and DCF and they turned out to be quite large (e.g. 3555 for DCF and 11937 for JEF for perfective changes) respectively. Hence, we decided not to use T-tests to statistically test our hypotheses, and present the results with histograms. The following is a summary and possible explanation of the results from our analysis of software changes for JEF and DCF.

RQ1: Does the distribution of change types vary for different development characteristics? We plotted our data in a histogram, shown in Fig. 1. From Fig. 1-a) we observed the following for JEF:

1) Decreasing perfective, corrective, preventive and adaptive changes over the three releases. The sudden drop in number of perfective changes for JEF between release 1 and release 2 and 3, yields that release 2 and 3 did not have much requirement changes and was based more on third party components. We can also see that there is not a big difference in the number of changes between release 2 and 3.
2) The preventive and adaptive changes decrease towards 0 between release 2 and release 3.
3) For the 3rd release the dominating changes are perfective and corrective, but the perfective changes are the most frequent ones.

For DCF (Fig. 1-b) we observed that:

1) Although the number of changes goes down for DCF between release 1 and 2 (before and after refactoring) for all change types, there is not a tendency that shows that any of these change types are decreasing.
2) It seems that corrective changes remain in the 25% of the changes.

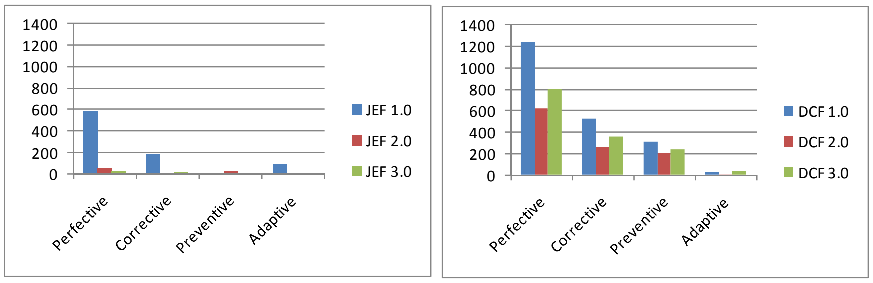

Fig. 1. Number of Changes: a) JEF, b) DCF.

Fig. 1 shows that *perfective and corrective changes* account for the majority of changes, for both the reusable JEF framework and the non-reusable DCF application. Our results confirm some of the findings from earlier studies (see Table 1), which shows that perfective and corrective changes are the most frequent ones independent of which kind of development characteristics the applications have. However, for JEF compared to DCF the adaptive changes follow closely. Regarding the perfective changes a contributing factor on DCF was an incomplete and poorly documented design, which required a high number of improvements over time. Important factor for JEF was to develop a common framework to support GUI (Graphical User Interface) development "up front" (developing without knowing all the functionalities a framework may need). The *least frequent changes* for the non-reusable application are the *adaptive changes*, and for the reusable framework the *least frequent changes* are the *preventive changes*. Contributing factors for the preventive and adaptive changes for DCF were:

- *Preventive changes*: Time pressure and incomplete and poorly documented design lead to some refactoring, since everything was not implemented optimally the first time. However, we can see a decrease in the preventive changes before (release 1) and after (release 2) refactoring.
- *Adaptive changes:* Minor changes were made to the environment/platform, which explains the small amount of *adaptive changes*.

Contributing factors for the preventive and adaptive changes for JEF were:

- *Preventive changes*: JEF did not go through the same time pressure as DCF during development. That resulted in a higher code quality for JEF, and less need for refactoring.
- *Adaptive changes*: StatoilHydro ASA changed their version control system from PVCS to Rational ClearCase in the middle of the project. All the files in the PVCS had a java comment, but when StatoilHydro ASA switched to Rational ClearCase the java comments in all the files were removed. The reason for why these changes are seen as adaptive changes is due to that these files had to be adapted to a different version control system (see section 2 for definition of adaptive changes). The higher frequency (compared to DCF) of adaptive changes can also be explained by the fact that JEF is built over various third party components, and changes in these components will cause changes in the framework.

We can see from Fig. 1 that JEF has a higher amount of adaptive changes than DCF. For JEF we see that adaptive changes accounted for more than usual compared to DCF, but still a fairly low number. This might be some surprising given that we expected JEF to need to be reused in a number of different environments/applications. However, this can partially be explained by the fact that the data we collected from Rational ClearCase includes just one application reusing the JEF framework. There are other application reusing JEF but they are for the time being under development and no data is available.

Answering our research questions:

- *RQ1.1: Does JEF have higher adaptive changes than DCF?* Yes, JEF (total number of changes 94) has higher adaptive changes than DCF (total number of changes 58).
- *RQ1.2: Is there a decrease in the preventive changes before and after refactoring for DCF?* Yes, there is a decrease in the preventive changes before (total number of changes 306) and after (203 for release 2 and 240 for release 3) refactoring for DCF. We can see there is a slightly increase between release 2 and 3 (18%), but still the number of changes are less for release 3 compared to before refactoring.
- *RQ.1.3: Do perfective and corrective changes account for the majority of the changes, with adaptive following closely?* Yes, perfective and corrective changes account for the majority of changes for JEF and DCF, but it is only for JEF that adaptive changes follow closely.

RQ2: What change types are the longest active for different development characteristics? From Fig. 1-a) we saw there was not a big difference in the number of changes between release 2 and 3. Therefore, we decided not to divide the JEF framework into three releases for our analysis of RQ2, since it will not affect the average. This means that for RQ2 we will here compare DCF release 1, 2 and 3 against the whole JEF framework.

By comparing the change types that are longest active for JEF and DCF, we found from Fig. 2-a) that *adaptive* (average of 50,2days) changes are longest active for JEF. This is because StatoilHydro ASA changed their version control system from PVCS to Rational ClearCase in the middle of the project. All the files in the PVCS had a java comment related to this version control system, but when StatoilHydro ASA switched to Rational ClearCase the java comments in all the files were removed. The JEF framework is built over various third party components, and changes in these components will cause changes in the framework. However, we can speculate that adaptive changes were longest active for JEF, because they affected many files. Another reason could be that adaptive changes were given low priority to fix. Thus, these files may have been checked out while developers might have been busy with other tasks with higher priority.

From Fig. 2-b) we can see that *preventive* changes (average of 17,0 days) are longest active for DCF, and the number of days for preventive changes drops (84% in average) between the two first releases of DCF. This is because before refactoring the code was difficult and hard to maintain (release 1), but after the refactoring the code became easier to maintain (release 2).

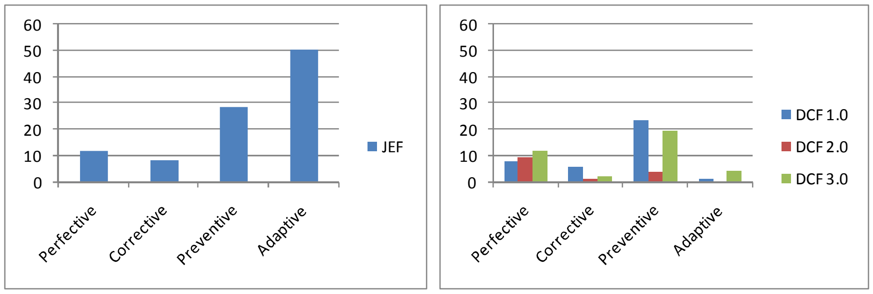

Fig. 2. Average #of days the changes are active: a) JEF, b) DCF.

It is important to clarify that the changes that are longest active do not mean that they require more effort, since we do not have the effort data. However, by looking into what change types are active longest we might to some extant be able to say if these changes stays longer in the applications and require more time to fix.

Answering our research questions:

- *RQ2.1: Are adaptive changes longer active in JEF than DCF?* Yes, adaptive changes are longer active for JEF (average of 50,2 days) than DCF (average of 2,5 days).
- *RQ2.2: Are preventive changes longer active before refactoring than after for DCF?* Yes, preventive changes are longer active before refactoring; release 1 has an average of 23, 5 days. While after refactoring; release 2 has an average of 3,8 days, and release 3 has an average of 19, 3 days. We can see there is an increase between release 2 and 3 (80% in average), but still the average number of days are less for release 3 compared to before refactoring.

RQ3: How localized are the effects of different types of changes, for different development characteristics? For RQ3 we will also compare DCF release 1, 2 and 3 against the whole JEF framework. By comparing the average number of files changed for each change type (Fig. 3), we found that DCF has higher average amount of files modified for the preventive changes (14,5). From Fig. 3-a) we can see that JEF has higher amount of files changed for the adaptive changes (5,5).

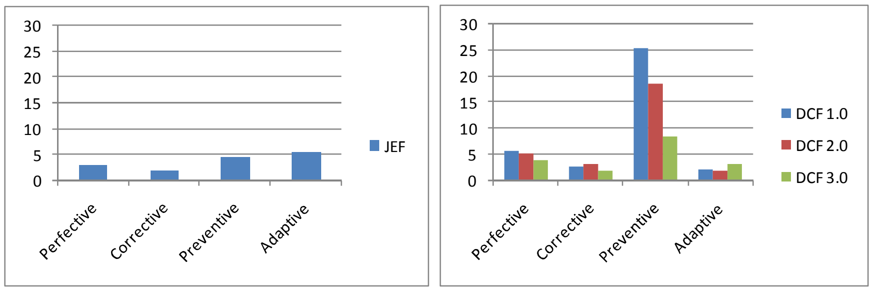

Fig. 3. Average amount of files modified: a) JEF, b) DCF.

From Fig. 3-b) we can also see the affect of the refactoring that happened between all the three releases, since the average number of files modified decreases. This decrease in the files for the preventive changes is related to adapting to an open source system framework to improve and ease the code related to handling GUI events. Before refactoring most of the code was developed by the developers and just some parts of the open source system framework were used. This made the code more complex, and difficult to maintain. Due to the high time-pressure the code was developed quickly and was defect-prone. However, during the refactoring the developers adapted more of the open source system framework and the code became much more structured.

Answering our research questions:

- o *RQ3.1: Is the average number of files modified for adaptive changes higher in JEF than DCF?* Yes, the average number of files modified for adaptive changes is higher for JEF (5,5 files modified) than DCF (2,4 files modified).
- o *RQ3.2: Is the average number of files modified for preventive changes higher before refactoring than after for DCF?* Yes, DCF (before refactoring) has in average 25,5 modified files. While DCF (after refactoring) has in average 18,5 modified files (release 2), and 8,4 modified files (release 3).

RQ2 combined with RQ3, we see the following results for DCF:

- o Even though the average number of days the changes are active are high for perfective and preventive changes, the number of files modified (within these two change types) are getting less over the three releases.

7 Threats to validity

We here discuss possible threats to validity in our case study and the steps we took to guard against them, using the definitions provided by [27]:

Construct Validity: All our data is from the pre- and post-delivery software changes (covering any reported changes) for the three releases of the reusable framework, and for the three releases of the DCF application.

External Validity: The object of study is a framework consisting of only seven components, and only one application. The whole data set of software changes in StatoilHydro ASA has been collected for three releases of the reusable framework, as well as for three releases of the application. So, our results should be relevant and valid for other releases of the framework and other future releases of the application. The entire data set is taken from one company. Application of these results to other environments needs to be considered on a case by case basis, considering factors such as:

- *The nature of the software development*: The DCF application and the JEF framework in our study are based on the object-oriented programming language, namely Java. Additionally, DCF is based on a waterfall process while JEF is based on a combined incremental/waterfall process.
- *The profile of the company and projects:* The profile of the company is an oil and gas company, and hence the projects are related to oil and gas field.

- *The way that software changes are defined and classified:* Our definition of software changes and other definitions used (see section 2), vary among the different studies.
- *The way that software changes are collected and measured:* We have collected software changes related only to the non-commented source lines of code for a reusable framework and a non-reusable application.

Internal Validity: All of the software changes for JEF and DCF were classified manually. Two researchers classified independently all the changes, and then cross-validated the results. This is to enhance the validity of the data collection process. A threat to the internal validity is the number of files we have selected from Rational ClearCase. However, we have 422 files for the JEF framework and 932 files for the DCF application, which should be enough files to draw a valid conclusion. We did a semi-random sampling to ensure the normal distribution between components.

Conclusion Validity: We verified the reasons for differences of software change profiles between the JEF and DCF by interviewing one senior developer (see section 5). Just asking one developer might cause systematic bias. However, we do not consider this possibility to be a threat for our investigation, because the senior developer has worked with both the JEF framework and the DCF application. His insights further supported our results for RQ1-RQ3.

8 Conclusion and Future work

Few published empirical studies characterize and compare the software changes for a reusable framework with those of a non-reusable application. We have presented the results from a case study for a reusable class framework and one application reusing it in StatoilHydro ASA. We studied the impact that software changes had on different development characteristics (e.g. impact of reuse and impact of refactoring). Our results support previous findings to the effect that perfective and corrective changes accounts for the majority of changes in both reusable and non-reusable software, but it is only for the reusable framework that adaptive changes follow closely. We also observed that DCF faced higher time-to-market pressure, more unstable requirements, and less quality control than the reusable framework.

When it comes to *designing for reuse* it does have an effect on the aspect of the change types. Our results indicate that adaptive changes have longer active time and files related to adaptive changes are more modified in JEF compared to DCF. The increase in adaptive change might be a result of successfully shielding the end user (i.e. DCF developer) from changes from the vendors. Additionally, preventive changes are more common in DCF (due to the refactoring that took place). So, the amount of changes, as well as the effect on the localization of changes will not be similar to the systems not necessarily designed for reuse.

Non-reusable applications usually face more unstable requirements, higher time-to-market pressure, and less quality control than the reusable framework. Therefore, their poorer quality is not surprising. So, making a component reusable will not automatically lead to better code quality. In order to lower the amount of software changes of the reusable component, it is important to define and implement a systematic reuse policy; such as better design [28] and better change management [21].

In addition, we have seen a positive affect for the *refactoring*. A system with poor structure initially has to deal with more frequent preventive changes before refactoring than after. However, our results indicated that there was an increase in preventive changes between release 2 and 3 (after refactoring), but the increase in release 3 was still less than before refactoring.

The lesson learned here is that developing a framework "up front" (developing without knowing all the functionalities a framework may need) is always difficult and challenging, since you do not know all of the requirements that will appear when a reusable framework is being used.

One interesting question raised by StatoilHydro ASA is whether the results of our study could be used as input to improve future reuse initiatives. In addition, we intend (i) to expand our dataset to include future releases of the JEF framework, future releases of the DCF application, and new applications (further reuse of the JEF framework), and (ii) to refine our research questions on the basis of the initial findings presented herein.

Acknowledgement

This work was financed by the Norwegian Research Council for the SEVO project [29] with contract number 159916/V30. We thank StatoilHydro ASA for involving us in their reuse projects. This work is also partially sponsored by NSF grant CCF0438933, "Flexible High Quality Design for Software."

References

1. Bennett, K.H et al.; Software Maintenance and Evolution: A Roadmap. In 22nd Intl. Conf. on Software Engineering, pp. 73-78. IEEE Press, Limerick (2000).
2. Lehman, M.M et al.; Programs, Life Cycles and Laws of Software Evolution. In. Proc. Special Issue Software Eng., IEEE CS Press, 68(9):1060-1076, 1980.
3. Postema, M. et al.; Including Practical Software Evolution in Software Engineering Education, IEEE Press, 2001.
4. Sommerville, I; Software Engineering, Sixth Edition, Addison-Wesley, 2001.
5. Bennet, P.L; Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations, Addison-Wesley Pub, 1980.
6. Lientz, B.P et al.; Characteristics of Application Software Maintenance. Communications of the ACM, 21(6): 466-471, 1978.
7. Mockus, A et al.; Identifying Reasons for Software Changes Using Historical Database. In Proc. IEEE Intl. Conf. on Software Maintenance, pp.120-130. IEEE CS Press, San Jose (2000).
8. Schach, S.R, et al.; Determining the Distribution of Maintenance Categories: Survey versus Management. In Empirical Software Engineering 8, pp. 351-366, December 2003.
9. Abran, A. et al.; Analysis of Maintenance Work Categories Through Measurement. In Proc. Conf on Software Maintetance, pp. 104-113. IEEE CS Press, Sorrento (1991).
10. Yip, S. et al.; A Software Maintenance Survey. In Proc. 1st Int. Asia- Pacific Software Engineering Conference, pp.70-79, Tokyo (1994).
11. Basili, V. et al.; Understanding and Predicting the Process of Software Maintenance Releases. In 18th Intl. Conf. on Software Engineering, pp. 464-474. IEEE CS Press, Berlin (1996).

12. Mohagheghi, P. et al.; An Empirical Study of Software Change: Origin, Impact, and Functional vs. Non-Functional Requirements. In Proc. at Intl. Symposium on Empirical Software Engineering, pp. 7-16. IEEE CS Press, Los Angeles (2004).
13. Gupta, A et al.; An Empirical Study of Software Changes in Statoil ASA – Origin, Piority Level and Relation to Component Size. In Intl. Conf. on Software Engineering Advances, pp.10. IEEE CS Press, Tahiti (2006).
14. Jørgensen, M.; The Quality of Questionnaire Based Software Maintenance Studies. ACM SIGSOFT – Software Engineering Notes, 1995, 20(1): 71-73.
15. Sousa, M. et al.; A Survey on the Software Maintenance Process. Intl. Conf. on Software Maintenance, pp. 265-274. IEEE CS Press, Bethesda (1998).
16. Satpathy, M et al.; Maintenance of Object Oriented Systems through Re-engineering: A Case Study. In Proceedings of the 10th Intl. Conf. on Software Maintenance, pp. 540-549. IEEE CS Press, Montreal (2002).
17. Lee, M.G et al.; An Empirical Study of Software Maintenance of a Web-based Java Application. In Proceedings of the IEEE Intl. Conf. on Software Maintenance, pp. 571-576. IEEE CS Press, Budapest (2005).
18. Evanco, M.; Analyzing Change Effort in Software During Development. In Proc. 6th Intl. Symposium on Software Metric, pp. 179-188, Boca Raton (1999).
19. Burch, E. et al.; Modeling Software Maintenance Requests: A Case Study. In Proceedings of the Intl. Conf. on Software Maintenance, pp. 40-47. IEEE CS Press, Bari (1997).
20. Mohagheghi, P. et al.; An Empirical Study of Software Reuse vs. Defect Density and Stability. In Proc. 26th Intl. Conf. on Software Engineering, pp. 282-292. IEEE-CS press, Edinburgh (2004).
21. Selby, W (2005). Enabling Reuse-Based Software Development of Large-Scale Systems. IEEE Transactions on Software Engineering. 31(6): 495-510.
22. Gupta, A. et al.; A Case Study of Defect-Density and Change-Density and their Progress over Time. In: 11th European Conf. on Software Maintenance and Reengineering, pp. 7-16. IEEE Computer Society, Amsterdam (2007). .
23. Zhang, W et al.; Reuse without compromising performance: industrial experience from RPG software product line for mobile devices. In Proc. 9th Intl. Software Product Line Conference, pp. 57-69. Springer, Rennes (2005).
24. Frakes, W.B. et al.; An industrial study of reuse, quality, and productivity. In Journal of System and Software, 2001, 57(2):99-106.
25. Algestam, H. et al.; Using Components to Increase Maintainability in a Large Telecommunication System. Proc 9th Int. Asia- Pacific Software Engineering Conference, pp.65-73, (2002).
26. Sampling calculator (http://www.macorr.com/ss_calculator.htm)
27. Wohlin, C, et al.; Experimentation in Software Engineering – An Introduction. Kluwer Academic Publishers, 2002.
28. Succi, G. et al.; Analysis of the Effects of Software Reuse on Customer Satisfaction in an RPG Environment. IEEE Transactions on Software Engineering, 2001, 27(5): 473-479.
29. Sevo project (http://www.idi.ntnu.no/grupper/su/sevo/)

P5: A Case Study Comparing Defect Profiles of a Reused Framework and of Applications Reusing it

Accepted with minor revisions (2 May 2008) to Journal of Empirical Software Engineering.

Anita Gupta[1], Jingyue Li[1], Reidar Conradi[1], Harald Rønneberg[2] and Einar Landre[2]

[1] Dep. of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU), Trondheim, Norway
{anitaash, jingyue, conradi}@idi.ntnu.no

[2] StatoilHydro ASA KTJ/IT, Forus, Stavanger
{haro, einla}@statoilhydro.com

Abstract The benefits of software reuse have been studied for many years. Several previous studies have observed that reused software has a lower defect density than newly built software. However, few studies have investigated empirically the reasons for this phenomenon. To date, we have only the common sense observation that as software is reused over time, the fixed defects will accumulate and will result in high-quality software. This paper reports on an industrial case study in a large Norwegian Oil and Gas company, involving a reused Java class framework and two applications that use that framework. We analyzed all trouble reports from the use of the framework and the applications according to the Orthogonal Defect Classification (ODC), followed by a qualitative Root Cause Analysis (RCA). The results reveal that the framework has a much lower defect density in total than one application and a slightly higher defect density than the other. In addition, the defect densities of the most severe defects of the reused framework are similar to those of the applications that are reusing it. The results of the ODC and RCA analyses reveal that systematic reuse (i.e. clearly defined and stable requirements, better design, hesitance to change, and solid testing) lead to lower defect densities of the functional-type defects in the reused framework than in applications that are reusing it. However, the different "nature" of the framework and the applications (e.g. interaction with other software, number and complexity of business logic, and functionality of the software) may confound the causal relationship between systematic reuse and the lower defect density of the reused software. Using the results of the study as a basis, we present an improved overall cause-effect model between systematic reuse and lower defect density that will facilitate further studies and implementations of software reuse.

Keywords *Software reuse, Software defect, Empirical study*

1 Introduction

Software reuse is a management strategy, where development *for* reuse refers to the deliberate development of software components that can be reused, and development *with* reuse refers to the inclusion of these reusable components in new and future software (Sindre et al. 1995). Since the 1970s, there has been a focus on how to develop software for/with reuse, technical/managerial/organizational aspects, measuring reuse in terms of quality and productivity, and reporting the success and failure of reuse practices. Several industrial empirical studies (Lim 1994; Mohagheghi 2004; Thomas 1997; Succi 2001; Selby 2005; Frakes 2001; Baldassarre 2005; Zhang 2005; Morad 2005) have concluded that reuse reduces the defect density and therefore helps to improve the quality of the system. A number of explanations for the lower defect density of the reused software have been proposed. For example, (i) reused software has been used by several different clients who have had defects fixed and the accumulated defect fixes will result in software of higher quality (Lim 1994); (ii) reused software will have better quality because few functions have been added to it (Thomas 1997; Frakes 2001; Selby 2005); and (iii) reuse-oriented software will be tested thoroughly before it is selected for reuse (Baldassarre 2005). However, few systematic explanatory studies have been performed to examine the decisive factors of the overall cause-effect relationship between systematic or ad hoc reuse and the lower defect density of reused software.

The purpose of this study is to compare the defect profile of a piece of software that is being reused and the software that is reusing it, and to find explanations for the possible similarities and differences between their defect profiles. We analyzed all defects introduced by developers (later detected either by testers or users) from trouble reports for all releases of a reused class framework, called Java Enterprise Framework (JEF), in the IT-department of a large Norwegian Oil & Gas company, as well as from two applications that were reusing the framework "as-is", namely Digital Cargo Files (DCF) and Shipment and Allocation (S&A).

We first compared the overall defect density (number of defects/non-commented source lines of code) for the reusable framework and the applications. Then we conducted an Orthogonal Defect Classification (ODC) analysis to compare the defect densities and severities of different defect types for the framework and the applications. After that, we studied the possible impacts that those defects would have on the user. Finally, we performed a Root Causal Analysis (RCA) to interpret our findings.

Our study supersedes previous studies (see Table 1) because we not only compared the overall defect density of a reused framework and the applications that are reusing it, but also classified the defects using ODC and compared the defect densities and severities of each defect type. In addition, the follow-up RCA attempted to explain why the reused framework has lower or higher defect densities of certain defect types, compared with those of the applications reusing it.

The results show that software reuse is helpful for reducing the number of defects, not only because it has been reused many times, but also because of the systematic reuse policy applied in the company, such as:

- well-defined requirements for the reusable framework,
- "characteristic" of the framework, such as looser coupling with other software that may be less complex, and
- cautious to incorporate changes to the reusable framework.

The first two factors will help to prevent defects. The third factor will help to prevent further defects from being introduced. This study therefore increases our understanding of the overall cause-effect relationship between systematic reuse and the possible lower defect density of the reused software, and reveals several decision-making factors that pertain to that relationship.

The paper is structured as follows. Section 2 presents related work, Section 3 presents the motivation for the research and the research questions. Section 4 describes the research design. Section 5 presents the results. Section 6 discusses the results. Section 7 concludes.

2 Related work

A systematic survey by Mohagheghi et al. (2007) summarized studies that have compared the defect densities of reused components with non-reused components, as shown in Table 1. Results from these studies show that continued reuse with slight modification results in significantly lower defect/problem density and significantly less effort expended on development and/or correction.

Table 1 Studies related to defect density and reuse

Quality focus	**Quality measures**	**Conclusion**
Reusable vs. non-reusable components (Lim 1994)	No definition of what a defect is. Defect density is given as defects/1000 non-comment source statements (KNCSS).	Reuse can provide improved quality, increased productivity, shortened time-to-market, and enhanced economics.
Reusable vs. non-reusable components (Mohagheghi et al. 2004)	Defect density (number of defects/lines of non-commented code)	-Reused components had lower defect density than those that were not reused. -Reused components had a higher number of defects of the highest severity before delivery, but fewer defects post-delivery.
Reusable vs. non-reusable components (Frakes et al. 2001)	Error density (number of errors per non-commented line of code) from the pre-delivery stage of the system.	More reuse results in lower error density.
Reusable vs. newly developed components (Thomas et al. 1997)	Error/defect densities (errors/defects per 1000 source statements). However, no definition of error/defect.	Reuse provides an improvement in error density (more than a 90% reduction) compared to new development.
Code reuse (Succi et al. 2001)	-Client complaint density (i.e. the ratio of client complaints to lines of code) -Defect density after the system is delivered to the client	Reuse is correlated significantly and positively with client satisfaction.
Reused, modified and newly developed modules (Selby 2005)	Module fault rate (number of faults in a module per non-commented source lines of code). Since an error correction may affect more than one module, each module affected by an error is counted as having a fault.	-Software modules reused without revision had the fewest faults, fewest faults per non-commented source line of code, and lowest fault correction effort. -Software modules reused with major revisions had the highest fault correction effort and highest fault isolation effort.

Some studies have proposed explanations for the lower defect density of reused components. For example, Lim (1994) proposed the following: 1) as work products are used multiple times, the defect fixes for each reuse accumulate, and gradually result in higher quality; and 2) more importantly, reuse provides incentives to prevent and remove defects earlier in the life cycle because the cost of prevention and debugging can be amortized over a greater number of uses. Succi et al. (2001) proposed that implementing a systematic reuse policy, such as the adoption of a domain-specific library, improves client satisfaction. Selby et al. (2005), Frakes et al. (2001), and Thomas et al. (1997) attributed the lower defect density of reused components to the smaller number, and lesser extent, of changes performed on them. In addition, Thomas et al. (1997) proposed the following: 1) if there is an expectation that components will be reused, it is more likely that they will be well-specified, particularly with respect to their reuse functionality; 2) the nature of the programming languages, i.e. FORTRAN and Ada in their cases, may affect the benefits of reuse; and 3) the experience with reuse in an organization and the approach taken towards reuse are likely to influence the nature of defects. A close examination of these studies illustrates that:

- Most studies compared only the number of defects between reused and non-reused components without going into further detail. The one exception is Thomas et al. (1997), who divided the defects into defect types and compared the number of defects of each type. However, no studies have so far investigated differences in defect densities in reused components with respect to the type of defect.
- Many factors may influence the success or failure of software reuse (Morisio et al. 2002, Rothenberger 2003), such as management commitment, the process by which reuse is

introduced, and human factors. It is therefore necessary to investigate which factors contribute positively to the lower defect density of reused software and which contribute negatively. In addition, it is important to understand which factors need to be excluded before analyzing the relationship between software reuse and lower defect densities of reused software. Some studies (Lim 1994; Succi et al. 2001; Selby et al. 2005; Frakes et al. 2001; Thomas et al. 1997) have attempted to attribute the lower defect densities of reused vs. non-reused software to the practices of reuse. However, few of them have done convincing cause-effect analyses. Most of them simply proposed possible explanations without providing confirmation, as shown in Fig.1.

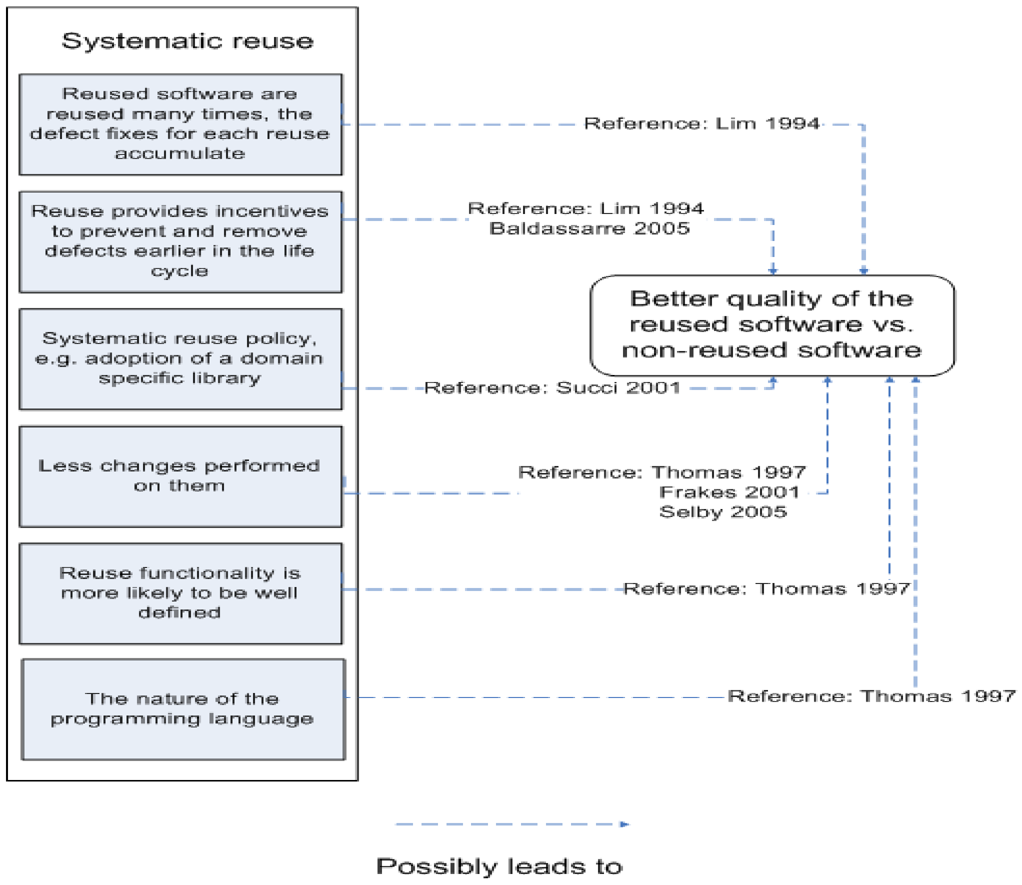
Systematic reuse
Reused software are reused many times, the defect fixes for each reuse accumulate
Reuse provides incentives to prevent and remove defects earlier in the life cycle
Systematic reuse policy, e.g. adoption of a domain specific library
Less changes performed on them
Reuse functionality is more likely to be well defined
The nature of the programming language
Reference: Lim 1994
Reference: Lim 1994
Baldassarre 2005
Better quality of the reused software vs. non-reused software
Reference: Succi 2001
Reference: Thomas 1997
Frakes 2001
Selby 2005
Reference: Thomas 1997
Reference: Thomas 1997
Possibly leads to

Fig.1 Current research proposals regarding the overall cause-effect relationship between software reuse and the lower defect/error density of reused software

3 Research motivation and research questions

Knowledge of the factors that govern the relationship between software reuse and lower defect density will help industrial practitioners to implement more cost-effective software reuse practices. The acquisition of such knowledge will require a greater number of detailed empirical studies of industrial practices. The primary motivation of this study was to compare the density and severity of the defects in the reused software with those of the software that reuses it. A secondary motivation was to try to explain the possible similarities and differences of the defect densities in reusable software and software that reuses it. Thus, the research questions we addressed are:

RQ1: What is the overall defect density of reusable software vs. that of software that reuses it?

Studies shown in Table 1 indicate that reused software has a lower defect density than that of non-reusable software. RQ1 is designed to study whether the same trend will be discovered in our study.

RQ2: What is the density of specific types of defect in reusable software vs. that of software that reuses it?

Most studies shown in Table 1, except Thomas et al. (1997), did not investigate whether the defect densities of specific types of defect in reusable software are lower than those of non-reused software. The purpose of RQ2 is to investigate the issue raised by RQ1 more deeply, by classifying the defects into different types and comparing the defect density for each of them.

RQ3: What are the relative severities and most severe defects in reusable software vs. those in software that reuses it?

Lim (1994) found that the defects in reused software were more serious in pre-delivery than those in non-reused software. RQ3 investigates the relative severity of defects in reusable vs. non-reusable software. In addition, we will examine what types of defect are most severe for the reusable software vs. those of the software that reuses it.

RQ4: What impacts on the client do defects in reusable software have vs. those in software that reuses it?

The impact of a defect on the client refers to what the user notices or would notice, if the defect persists or would persist after the deployment of the application at the user's site.

4 Research design

We investigated three software systems from StatoilHydro ASA, which is a large Norwegian oil and gas company. In this section, we first introduce the company, the three systems, and trouble reports for these systems. We then illustrate how the trouble reports were analyzed and how the follow-up Root Causal Analyses were performed.

4.1 Data collection

4.1.1 The company

StatoilHydro ASA has.a total of about 31,000 employees, with its headquarters in Norway and branches in 40 countries. The IT department of the company is responsible for developing and delivering domain-specific software, to give key business areas better flexibility and efficiency in their regular operations. It is also responsible for the operation and support of mass IT systems. This department consists of approximately 100 developers, located mainly in Norway. In addition, StatoilHydro ASA subcontracts a great deal of software development and operations to consulting (software) companies.

4.1.2 The investigated systems

We investigated three systems. One is a reusable framework called JEF. The remaining two, which reuse JEF, are applications called DCF and S&A.

The company initiated their reuse strategy in 2003 with pre-studies. At that time, a reusable software framework was under development. This framework is based on J2EE (Java 2 Enterprise Edition), and is a Java technical framework for developing Enterprise Applications. Thus, the framework is called the "JEF framework" and consists of seven separate components. The latest release of JEF components contained a total of 20348 Non-commented Source Lines of Code (NSLOC), and can either be applied separately or together when developing applications. Table 2 shows the size and release date of the three JEF releases (excluding third-party components). JEF is designed on the basis of a technical architecture for all J2EE systems in the company. This architecture has four logical layers, as follows (from top to bottom):

(1) Presentation: responsible for displaying information to the end-user and to interpret end-user input.

(2) Process: provides support for the intended tasks of the software, and configures the domain objects.

(3) Domain: responsible for representing the concepts of the business, and information about the business and business rules. This layer is the heart of the system.

(4) Infrastructure: provides generic technical services, such as transactions, messaging, and persistence.

DCF is used mainly for document storage. It imposes a certain structure on the documents stored in the application. It assumes that the core part of the documents is based on cargo (load) and deal (contract agreement) data, as well as auxiliary documents pertaining to this information. DCF is meant to replace the current handling of cargo files, which are physical folders that contain printouts of documents that pertain to a particular cargo or deal. A “cargo file” is a container for working documents that are related to a deal or cargo that are used by all parties in the oil sales, trading, and supply strategy plan of the company. There are three releases of the DCF application. Table 3 gives an overview of the size and release date of the three releases (excluding the code of JEF and other third-party components).

S&A is an application that employs common business principles to enable efficiency and control in business processes that pertain to lift and cargo planning. Lift planning is based on a lifting program that generates an overview of the cargoes that are scheduled to be lifted. The lifting program operates on a long-term basis (e.g. 1 - 12 months), and generates tentative cargoes based mainly on closing stock and predicted levels of production. The lifting program is distributed to the partners (other oil and gas companies, such as Shell and Gaz de France), so that they can plan the lifting of their stock. The planning of shipment and cargo covers activities to accomplish such lifting. Input to the process is the lifting program. Users use the lifting program to enter detailed information about a cargo, based on documented instructions from partners, and perform short-term planning based on the pier capacity and storage capacity. After loading, sailing telex and cargo documents are issued. Then the cargo is closed and verified. The S&A application allows the operators to carry out “what-if” analysis on shipments that are to be loaded at terminals and offshore. The current trading system (“SPORT”) is not able to handle complex agreements (i.e. the mixing of oil of different qualities within the same shipment), or automating the transfer and entry of related data (which is currently often done manually). The main goal of the S&A application is to replace some of the current processes/systems, as well as to offer some new functionality. The S&A application has also three releases. Table 4 gives an overview of the size and release date of these releases (excluding the code of JEF and other third-party components).

Table 2 Size and release date of the three JEF releases

Release 1: 14. June 2005	**Release 2: 9. Sept. 2005**	**Release 3: 8. Nov. 2005**
16 875 NSLOC	18 599 NSLOC	20 348 NSLOC

Table 3 Size and release date of the three DCF releases

Release 1: 1. Aug. 2005	**Release 2: 14. Nov. 2005**	**Release 3: 8. May 2006**
20 702 NSLOC	21 459 NSLOC	25 079 NSLOC

Table 4 Size and release date of the three S&A releases

Release 1: 2. May 2006	**Release 2: 6. Feb. 2007**	**Release 3: 12. Dec. 2007**
29957 NSLOC	50879 NSLOC	64319 NSLOC

From Tables 2, 3, and 4 we can see that the framework and the applications are growing. JEF consist of seven components. These are being used in PDM (Physical Deal Maintenance) and reused in DCF and S&A. However, DCF and S&A are not being used in any other applications. JEF is a framework that is reused in DCF and S&A and in other projects “as-is”. This is how we can say that JEF is reusable, and DCF and S&A are non-reusable. JEF, DCF, and S&A will grow in size because when the clients use the applications they will make some changes to it, which will also require changes to the framework. For instance, adding new functionality to the reusable and non-reusable software will result in growth for JEF, DCF, and S&A. Another explanation of the growth of the framework and the applications is that when a defect is found in Release 1 the fixes will be included in Release 2, etc. Thus, the framework and the application will grow.

JEF Release 1 was finished around June 2005, and PDM in the summer 2005 was the first application to use the JEF framework (Release 1). In this period, some weaknesses in the framework were discovered. These changes were then incorporated into JEF, ending early September 2005. Then, Release 2 of the JEF framework was delivered. The DCF application reused Release 2 of the JEF framework during late summer and autumn 2005. After DCF reused the JEF framework, some more minor changes were made to the framework, which were finished by early November 2005. Then, Release 3 of the JEF framework was deployed. The second application, S&A, reused Release 3 of the JEF framework, and was developed during early 2006. The relation between the JEF and applications using/reusing it are shown in Fig. 2. The company uses the same test team and has the same test coverage for both the reusable and non-reusable software. For instance, for unit testing, 85% of the code lines were executed by unit tests to ensure that the code worked as expected. However, detailed investigation of software testing lies beyond the scope of this paper and will be the topic of future work (see section 7). We have not included defects in the PDM application other than those in JEF in our study, because PDM was the first application to *use* JEF, not *reuse* it (like DCF and S&A).

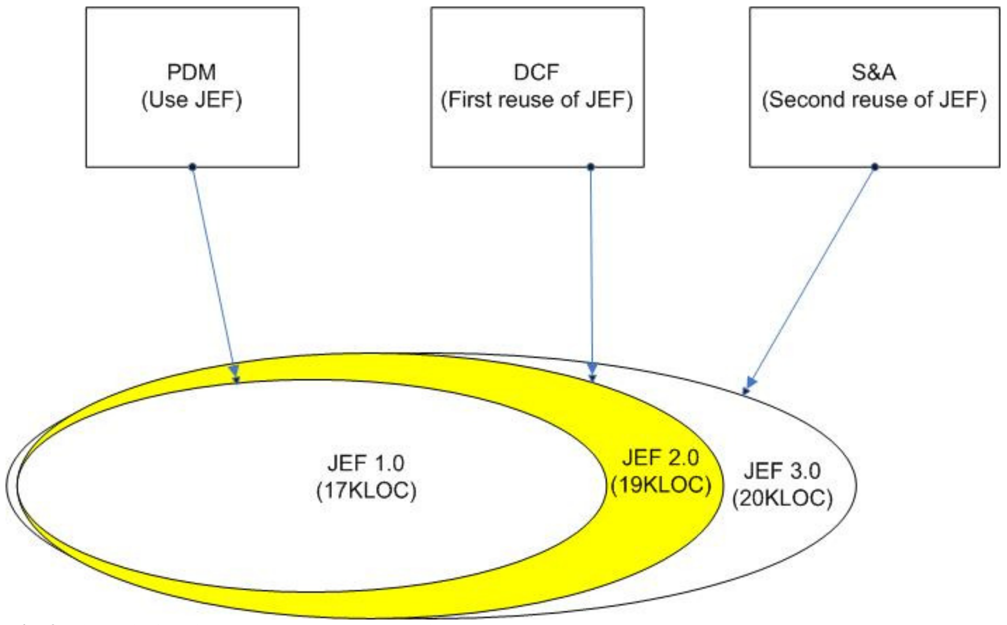

Fig.2 The relation between JEF, DCF, and S&A

4.1.3 The investigated trouble reports

When a defect is detected during integration/system testing and all field use, a trouble report is written and stored in the Rational ClearQuest tool. Therefore, the trouble reports include all defects introduced by developers and detected in pre-delivery or post-delivery releases of the systems. All registered trouble reports can be exported as Microsoft Excel files. Each trouble report contains the following items:

- ID.
- Headline description.
- Priority (which indicates how urgent fixing a problem is) assigned by developers or testers:
    - *Critical* - means that the system does not fulfill critical business functionality or will disrupt other systems.
    - *High* - loss of a part of the required functionality or quality.
    - *Medium* - part of the required functionality or quality is lost, but that there are ways to work around the problem.
    - *Low* - defect has no important effect on the functionality or quality).
- Severity (which indicates how serious the problem is) as assessed by developers:
    - Critical, High, Medium or Low.
- Classification:
    - Error,
    - Error in other system,
    - Duplicate,
    - Rejected,
    - Postponed, and so on.
- Estimated effort to fix.
- Remaining time to fix.
- Subsystem location (e.g. one specific component of a system).
- System location (e.g. JEF, DCF, or S&A).
- Updated action and timestamp record for each new state that the defect enters in the workflow.

4.2 Data analysis

The data was analyzed in two stages. In the first stage, we analyzed the trouble reports of JEF, DCF, and S&A to answer the research questions RQ1 to RQ4 as follows:

- For RQ1, we divided the NSLOC of each system by the number of defects to calculate the defect density. The NSLOC was counted using the Eclipse tool, because that is the development tool used in the company.
- For RQ2, we first classified the defects of each system using defect types from a slightly modified Orthogonal Defect Classification (ODC) scheme from IBM (Chillarege et al. 1992). The attribute "defect type" captures the correction to resolve the defect. For example, defects of type "function" are those that require a formal design change. Detailed explanations of ODC and the definitions of defect types used in this study may be found in Appendix A. After the defects were classified, we divided the number of defects of each defect type by the total NSLOC of the corresponding system to get the corresponding defect density of each defect type.
- For RQ3, we first counted the number of defects of different severities in each system. We then divided the number of defects of different severities by the NSLOC. However, 25% of the severity data for DCF and JEF in ClearQuest was missing or incomplete for some of the defects. By contrast, the priority data for the defects were complete. We did a Spearman correlation test with SPSS 14.0 and found that the priority data correlates well with the severity data. For both DCF and JEF, the severities and priorities (i.e. for the 75% of defects for which complete priority and severity data was available) are significantly correlated (with p-value is less than .001) with a correlation coefficient more than .80. For S&A, the correlation coefficient between priority and severity is .90 (with p-value is less than .001). Therefore, we decided to use the priority data for the severity analysis in JEF, DCF, and S&A.
- For RQ4, we first classified the impact of each defect using the impact attribute of ODC (Chillarege et al. 1992). The definitions of different "impact" attributes used in this study are shown in Appendix B. Then, we divided the number of different impacts of defects by the NSLOC.

In the second stage of the data analysis, we performed a fish-bone Root Causal Analysis (Card 1998) by interviewing a senior developer who was familiar with development of both the JEF framework and the applications. We first showed him the results of our data analysis (to avoid a possible threat to validities of our results, we did not inform him of our research questions). We then asked him to interpret the causes of defects with respect to tools and environment, input and requirements, method, and people (Card 1998).

5 Results of the research questions and interpretations of the results

5.1 Collected trouble reports

Over all releases, there were 232 trouble reports for JEF, 592 for DCF, and 723 for S&A. Given that the defect type captures the attempt that was made to resolve the defect, we can only use those defects where the handling of the defect was complete and closed. Therefore, we included only complete and closed defects.

Table 5 gives an overview of the defects that were excluded. After excluding all the defects that were not complete and closed, 223 trouble reports remained for JEF, 438 for DCF, and 649 for S&A. We then classified these defects manually. The first and the second author of the paper classified all the defects separately and then compared the results jointly. During the classification and comparison, we noticed that some of the defects were classified as "not fault". We excluded these from our analysis: one from JEF, 13 from DCF, and two from S&A. So, in our data analysis, we used 222 defects for the three JEF releases, 425 defects for DCF, and 647 defects for S&A.

Table 5 Number of defects excluded in the analysis

	Defect states	**#Defects excluded from JEF**	**#Defects excluded from DCF**	**#Defects excluded from S&A**
Rejected	Rejected (developers not sure whether the defect is a defect)	1	67	26
Not solved	Postponed (defect postponed to later releases)	0	22	5
	Submitted (a defect is submitted, but without correction handling)	0	13	23
	Analysed (a defect is being analyzed)	0	4	4
	Assigned (a defect has been assigned to a tester)	3	11	5
	In progress (analysis of a defect is in progress)	3	2	8
Duplicate	Duplicate (duplicate of another defect)	2	35	3
Not fault	Not fault	1	13	2
	In total:	10 excluded (4%)	167 excluded (28%)	76 excluded (11%)
	Total defects analysed:	223	438	649

5.2 Answers to research questions

RQ1: What is the overall defect density of reusable software vs. that of software that reuses it?

The defect density of the JEF framework was 222/20 Kilo NSLOC=11.1 per Kilo NSLOC. The defect density of DCF was 425/25 Kilo NSLOC=17 per Kilo NSLOC. The defect density of S&A was 647/64 Kilo NSLOC = 10.1 per Kilo NSLOC. The results show that the JEF has a lower defect density than the DCF, but a slightly higher defect density than S&A.

RQ2: What is the density of specific types of defect in reusable software vs. that of software that reuses it?

By comparing the defects per Kilo NSLOC of the different defect types, as shown in Fig. 3, we found that the DCF application has a much higher defect density than the JEF with respect to four types of defect: relationship, function, data, and checking. The root cause analysis yielded by discussion with the senior developer showed that:

1) The DCF has a higher *relationship*-type defect density than the JEF because it is tightly coupled with several other applications in the company. By contrast, the coupling between the JEF and the other applications is looser.

2) There are three reasons why DCF has a higher *function*-type defect density than the JEF:

(i) The goals and requirements for the JEF were clearer and more stable than for the DCF. Although the DCF was based on the waterfall process, major changes to the requirements and new decisions were incorporated in late phases of the project. The development of the DCF suffered from more time pressure than the JEF.

(ii) In the DCF, the design specification was incomplete and missing. The developers did not have a detailed design at the beginning, and a lot of changes were made regarding functionality and design during the implementation period. The JEF had good documentation and therefore did not suffer from these problems.

(iii) The JEF did not experience major changes in the project phase. By contrast, work on the DCF was stopped for a while during the implementation phase to discuss and incorporate major changes.

3) The DCF has a higher *checking*-type defect density because it is primarily a business application, and has more rules and business logic. The same also is true for the *data*-type defect density.

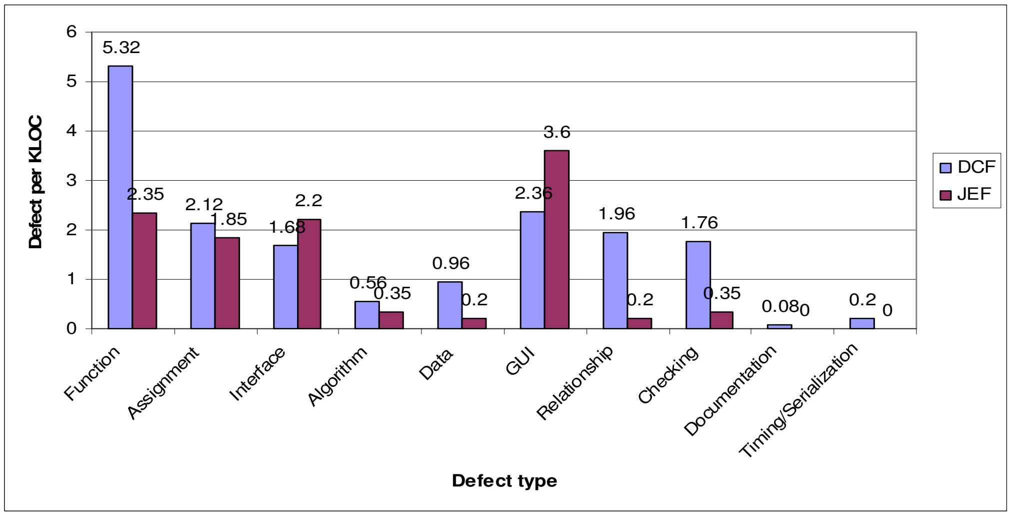

Fig. 3 Defect density for the different types of defect in the JEF vs. the DCF

By comparing the defects per Kilo NSLOC of the different types of defect, as shown in Fig. 4, we found that the S&A application has a much higher defect density than the JEF with respect to four types of defect: function, data, checking, and algorithm. The root cause analysis revealed that:

1) The S&A has a higher *function*-type defect density than the JEF because S&A consists of many user interfaces, and the users were rarely involved during the design and implementation of these interfaces. In addition, few developers with sufficient knowledge of the usability of the application were involved in the project. When the users had the chance to see the application, it became apparent that a lot of changes regarding functionality and design of the user interface needed to be made to satisfy the users' requirements.

2) The S&A has a higher *algorithm*-type defect density because of its complex business logic. One of the major parts of the S&A application is to do lift and cargo planning. This function is designed and implemented on the basis of a total analysis of the cargoes that are scheduled to be lifted (e.g. calculating which partners will lift the cargo and when), as well as the activities to accomplish the required lifting. Such lift and cargo planning requires a great deal of calculation. Hence, S&A, compared to the JEF, has implemented heavier algorithms to perform these calculations efficiently and properly.

3) The S&A has a higher *checking*-type defect density because it is primarily a business application (just as the DCF is), and therefore has more rules and business logic. The same goes for the *data*-type defect density.

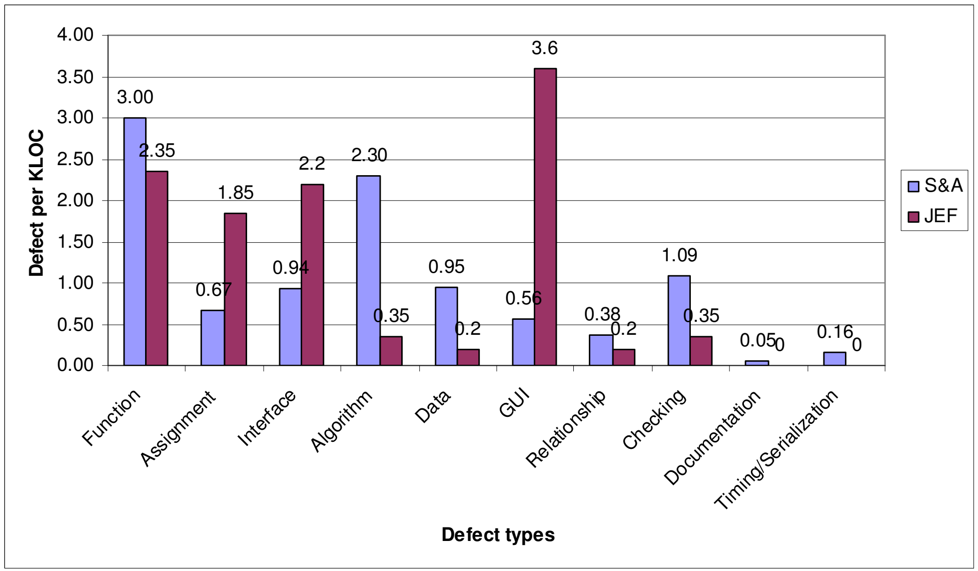

Fig.4 Defect density for the different types of defect in the JEF vs. the S&A

Results shown in Fig. 3 and Fig. 4 illustrate that the JEF has a higher density of interface-type defects than both the DCF and S&A. The root cause analysis reveals that the JEF has been used/reused by three applications, with the result that the component interfaces of JEF gradually needed to be corrected or improved to make the reuse easier and more efficient. In addition, the results presented in Fig. 3 and Fig. 4 show that the JEF has a higher defect density of GUI-type defects, simply because JEF has many more GUIs than the DCF and S&A and so there are more requests to alter the layout of some of the JEF GUIs, especially concerning data displays, buttons, and checklists.

RQ3: What are the relative severities and most severe defects in reusable software vs. those in software that reuses it?

The defect densities of defects with different severities are shown in Fig. 5. The results reveal that the JEF framework and the applications have almost similar defect densities for defects of *Critical* and *High* severity. To investigate whether these systems have similar profiles for the most severe defects, we analyzed the defect-type distributions of defects with different severities. The results for the JEF, DCF, and S&A are shown in Fig. 6, Fig. 7, and Fig. 8, respectively.

Fig. 6 shows that for the JEF framework, the types of defect that are of Critical and High severity are *interface* and *assignment*. The developers explained that the JEF is designed as a framework and its interface-type defects will affect many applications. Thus, the interface-type defects are usually given a high priority. The assignment-type defects usually have serious consequences, which may result in the JEF not being able to run properly.

Fig. 7 shows that for the DCF application, the types of defect that are of Critical and High severity are *relationship* and *function*. The DCF application has a close coupling with several other applications in the company. Therefore, these two types of defect are given high priority, because they indicate that the whole system will not perform as expected.

Fig. 8 shows that for the S&A application, the types of defect that are of Critical and High severity are *algorithm* and *function*. The S&A application has several algorithm-type defects due to all the calculations for lift and cargo planning. Thus, algorithm-type defects were regarded as severe. The function-type defect can be explained by missing functionality in the GUIs for the application. These two types of defect are given high priority because they indicate that the whole system will not perform as expected.

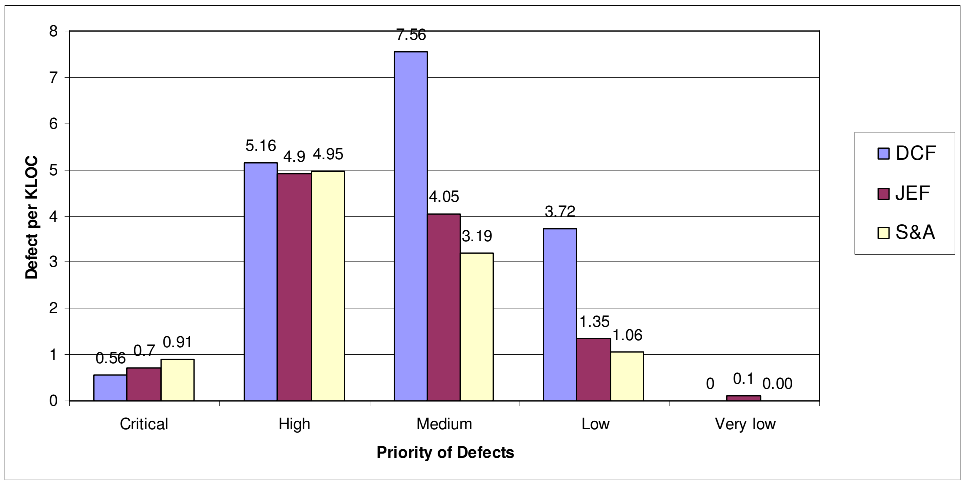

Fig.5 Defect density for defects with different severities (JEF vs. DCF vs. S&A)

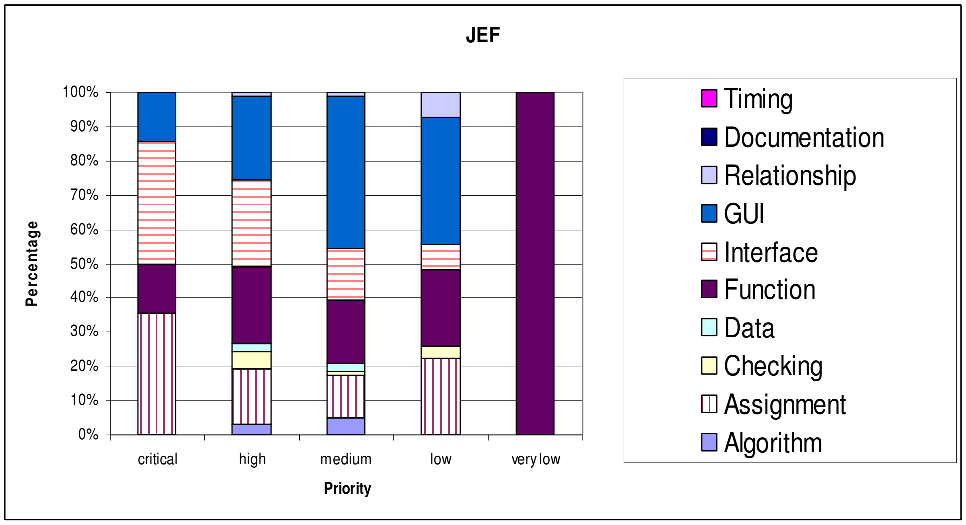

Fig.6 Distributions of different defects with different severities for JEF

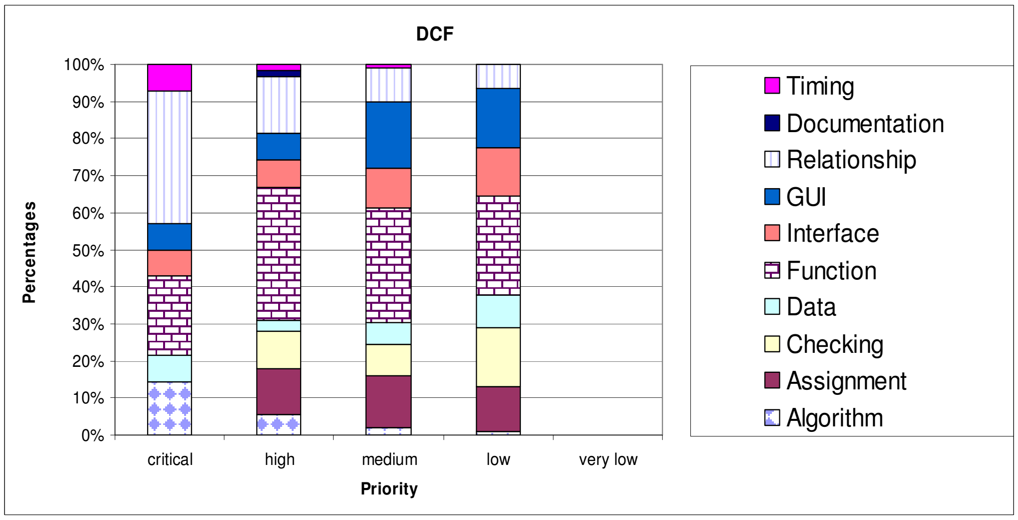

Fig.7 Distributions of different defects with different severities for DCF

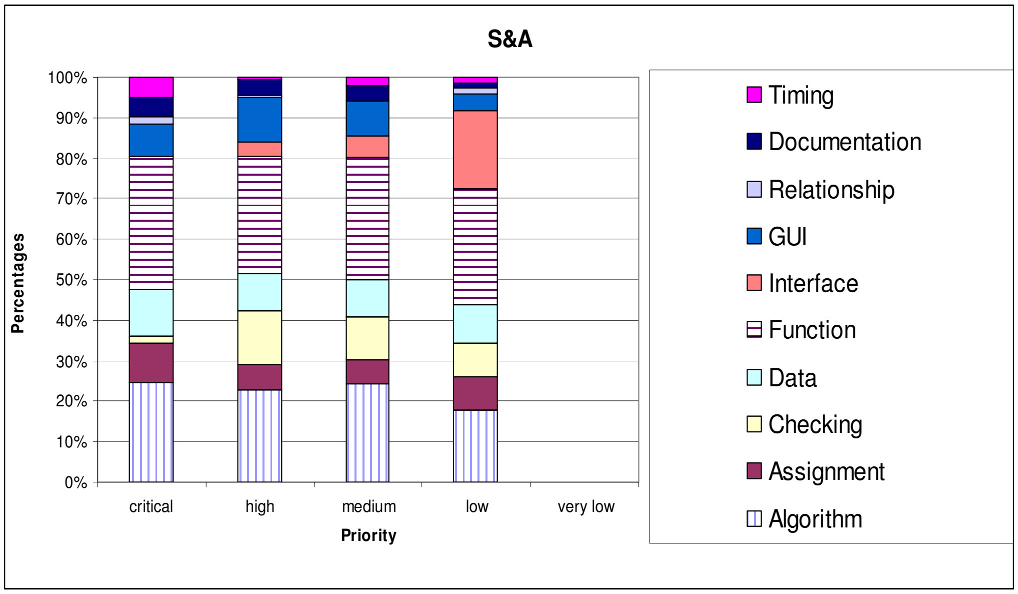

Fig.8 Distributions of different defects with different severities for S&A

RQ4: What impacts on the client do defects in reusable software have vs. those in software that reuses it?

The impact of defects in the JEF framework and the two applications are shown in Fig. 9. The results illustrate that impacts on *capability* and *usability* are the most common in all three systems. However, defects in the JEF have much less impact on capability than the two applications that reuse it. The developer explained that the DCF application had missing/incomplete functionality and unclear requirements from the beginning, which will mainly affect the capability. The users were not much involved in the implementation of the S&A application. When the users had the chance to see the application, many changes had to be made to satisfy the users' revised requirements. By contrast, the requirements for the JEF were much better defined at the beginning than for the DCF and S&A, which helped to diminish the defects' impact on the capability of the system. Given that the JEF, DCF and S&A all have a large amount of GUIs, it is not surprising that many defects will affect the usability of the system.

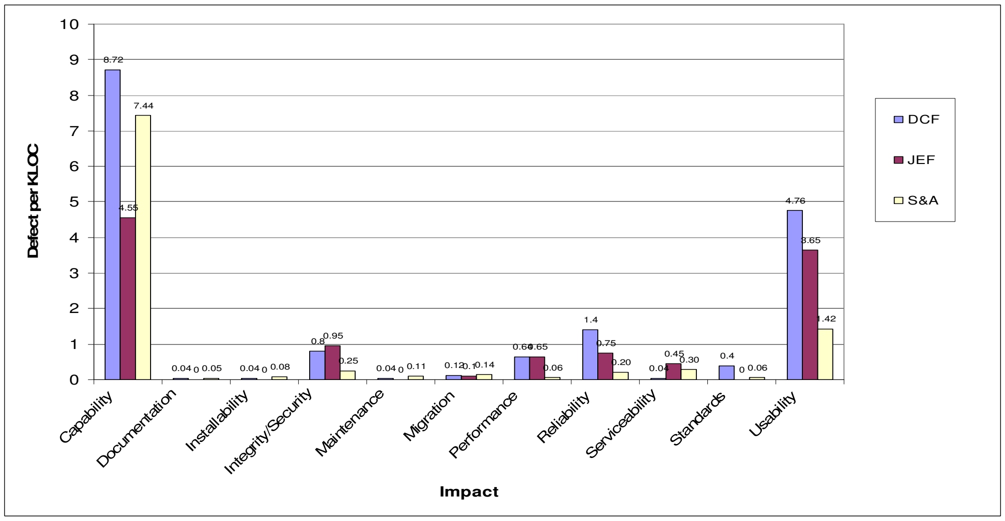

Fig.9 Impacts of defects for JEF, DCF, and S&A

Table 6 presents a summary of our results, along with the corresponding research questions.

Table 6 Summary of the results

Research questions	**Answers to research questions**		
	JEF (reused software)	DCF (reusing JEF)	S&A (reusing JEF)
RQ1: Defect density	11.1 per Kilo NSLOC	17 per Kilo NSLOC	10.1 per Kilo NSLOC
RQ2: Defect types with the highest density	Interface-type and GUI-type (JEF vs. DCF vs. S&A)	Function-type, relationship-type, checking-type, and data-type (JEF vs. DCF)	Function-type, data-type, checking-type, and algorithm-type (JEF vs. S&A)
RQ3: Most severe defect types	Interface-type and assignment-type	Relationship-type and function-type	Algorithm-type and function-type
RQ4: Most common impact of defects	Capability and usability	Capability and usability	Capability and usability

6 Discussion

6.1 Comparison with related work

Our results support some of the observations of the studies shown in Table 1 and Fig. 1, and contradict others. We have summarized the comparison of our results with previous studies in Table 7. Although we cannot deny the observations that reused software is reused many times and that the defect fixes for each reuse accumulate (Lim 1994; Baldassarre 2005), our data show that reused software may not have a lower defect density than non-reused software. Furthermore, software reuse will probably not reduce the density of the most severe defects either. The aspects of systematic software reuse that have helped to reduce the defect density of reused software are: well-designed functionality, solid design and testing, as well as cautions to changes. It is possible that the differences in content/focus (domain, functionality, and complexity) between reused software and non-reused software may confound the cause-effect relationship between reuse and lower defect density of the reused software. Using the results of this study as a basis, we revised the explanatory model of the overall cause-effect relationship between software reuse and the lower defect density of reused software that was presented in Fig. 1 into the model shown in Fig. 10.

Table 7 Comparison of results of previous studies with results of this study

Observations/conclusions from previous studies	**References**	**Evidence from this study**	**Conclusion of this study**	**Support/Against previous conclusions**
Reuse reduces the defect density and therefore helps to improve the quality of the system.	Mohagheghi et al. 2004; Selby 2005; Thomas et al. 1997; Succi et al. 2001.	Results for *RQ1* show that JEF has a lower defect density than DCF However, JEF has a slightly higher defect density than S&A.	Reuse will not necessary reduce the defect density.	Partly against.
Defects in reused components are more serious in pre-delivery than components that are not reused.	Lim 1994.	Results for *RQ3* show that JEF has a density of most severe defects similar to those of DCF and S&A.	Reused software has a density of the most severe defects similar to non-reused software.	Against.
Implementing a systematic reuse policy, such as adopting a domain specific library, improves client satisfaction.	Succi et al. 2001.	Results for *RQ2* show that JEF has been better designed and tested than DCF and S&A.	Systematic reuse policy helped to reduce the defect density of software to be reused.	Partly support (However, we did not measure client satisfaction; only the defect density).
Software modules without revision had the fewest defects.	Thomas 1997; Frakes 2001; Selby 2005.	Results for *RQ2* show that the company was more cautious when changing JEF than DCF and S&A.	Fewer changes helped to reduce the defect density.	In favour.
Reuse functionality is more likely to be well defined.	Thomas 1997.	Results for *RQ2* show that JEF has much lower defect density of functional-type defects than DCF and S&A. Results for *RQ4* show that defects of JEF have much lower impacts on capability than those of DCF and S&A.	Well-defined functionality of the reused software helped to reduce the defect density of the function defects.	In favour.
The nature of the programming language helped to reduce the defect density.	Thomas 1997.	Results for *RQ2* show that DCF and S&A have much higher defect densities of algorithm-type, data-type, and checking-type defects than JEF.	The domain and complexity differences between reused software and non-reused software will confound the differences in defect density.	In favour.

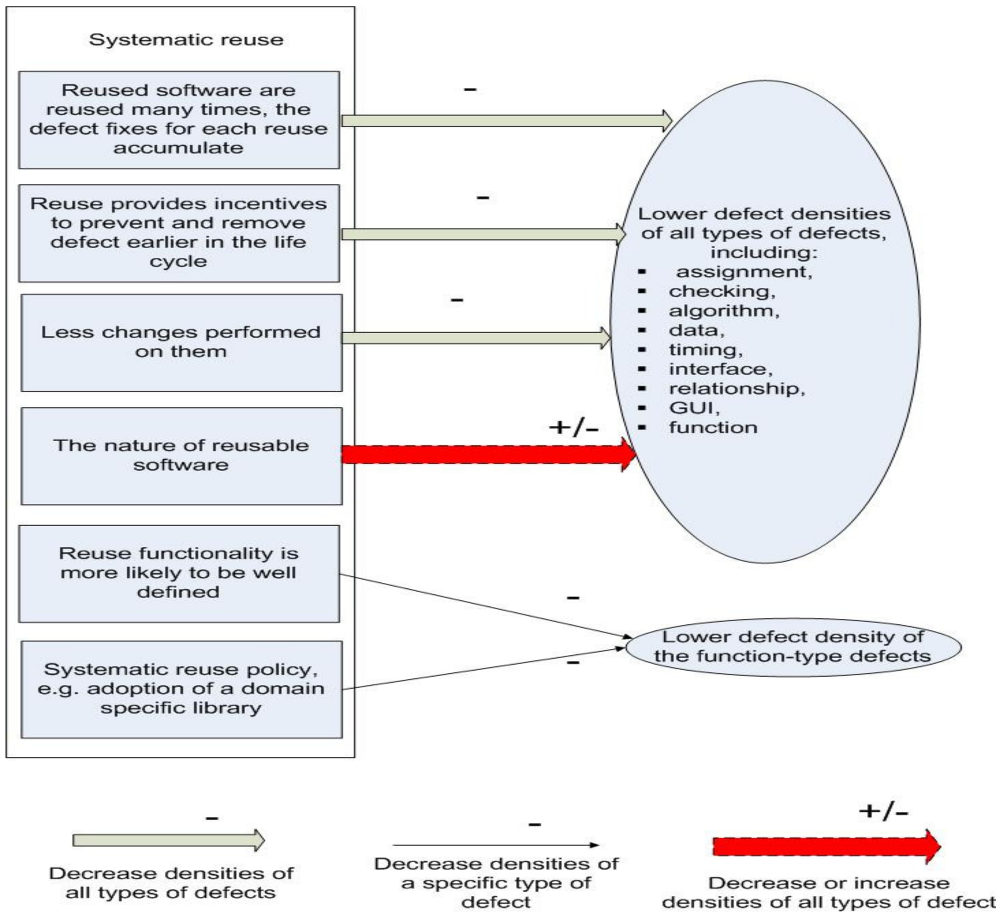
Systematic reuse
Reused software are reused many times, the defect fixes for each reuse accumulate
Reuse provides incentives to prevent and remove defect earlier in the life cycle
Less changes performed on them
The nature of reusable software
Reuse functionality is more likely to be well defined
Systematic reuse policy, e.g. adoption of a domain specific library
-
-
-
+/-
-
-
Lower defect densities of all types of defects, including:
assignment,
checking,
algorithm,
data,
timing,
interface,
relationship,
GUI,
function
Lower defect density of the function-type defects
-
Decrease densities of all types of defects
-
Decrease densities of a specific type of defect
+/-
Decrease or increase densities of all types of defect

Fig.10 Improved overall cause-effect model between software reuse and the defect densities of reused software

6.2 Recommendations to the IT industry on software reuse

By investigating the defect density of different defect types via ODC, the results for RQ2 show that the JEF and the two applications that are reusing it have different defect densities for different types of defect. The root cause analysis reveals that the lower defect density of the JEF is due partially to the systematic implementation of the reuse policy, such as clearly defined functionality, better design and testing (Succi et al. 2001), and better management of changes (Selby 2005). The higher defect densities of function-type defects in the DCF and S&A are due partially to higher time-to-market pressure, more unstable requirements, and less quality control. *Thus, it is important for industrial practitioners to define and implement a systematic reuse policy to improve the defect density of reusable software.*

The results for *RQ3* show that the most severe defects for the JEF are assignment-type and interface-type. This is because several other applications, e.g. DCF and S&A, need to use the functions of the reusable framework through its interface. Therefore, interface-type defects in the JEF may cause several of the applications that reuse the JEF framework to fail. *This indicates that more solid quality control or testing should be performed on reusable software to reduce the possible interface defects.*

Finally, some of the fields in the recorded defect data (e.g. defect severity) are incomplete. This indicates that the trouble reports have not been analyzed properly by the persons responsible and that little feedback has been given on the collected trouble reports. *If feedback were provided, the precision of data collection could be improved in the short run and promising changes to the process could be suggested in the long run.*

6.3 Threats to validity

We now discuss possible threats to validity in our case study, using the definitions provided by (Wohlin et al. 2002):

Construct Validity: Root Cause Analysis (RCA) is often performed on each defect (Leszak et al. 2000). One possible threat to construct validity is that we performed our RCA on a summary of all defects. Given that we did not perform a detailed analysis of each defect, we may have missed important causes of the defects. However, in StatoilHydro ASA several of the developers who were involved in the project are external consultants and when their work on the project was completed, they left. This made it difficult for us to trace defects back to each developer. Therefore, we did not have the resources to perform a root cause analysis of each defect. However, we selected 5% of defects at random for the JEF, DCF, and S&A, and performed a root cause analysis on each of these defects. The results support our conclusion for all research questions. In addition, we verified the reasons for differences in function-type defects (see section 6) between the JEF, DCF, and S&A by interviewing another senior developer. His insights are in line with the first senior developer with whom we discussed these reasons.

Internal Validity: All of the trouble reports for the JEF, DCF, and S&A were classified manually by us. The first and the second author of the paper classified all the defects separately and then cross-validated the results.

A threat to the internal validity is how the defects are reported at StatoilHydro ASA. Ambiguity could exist as to whether developers classify an incident as a trouble report or not. Due to the interaction between the JEF, DCF, and S&A, defects might have been attributed to the applications (DCF or S&A) that rightly should have been assigned to the framework (JEF); hence, the way in which defects are reported remains a threat.

Another threat to the internal validity is the incomplete and missing data on the severity of the defects reported for the JEF and DCF. We decided to use the priority data for the severity analysis in the JEF and DCF, which may constitute a threat to internal validity. However, we performed a Spearman correlation test and found that the severity data and priority data are correlated significantly.

The ideal thing would be to look at defects on the component level rather than the system level. However, the software systems investigated are large and complex, so one defect may affect several components. This complexity makes it difficult to classify which specific component a defect belongs to. Hence, we evaluated the whole system. If we had compared defects at the component level, there may have been more errors of misclassification, which would have constituted a more serious threat to internal validity.

External Validity: The entire data set was taken from one company. The object of study was a class framework, and only two applications. Generalization to similar contexts in other organizations should be discussed case by case.

Conclusion Validity: We performed our analysis on the basis of an initial collection of data. A possible threat to validity is that the differences among types of defect with respect to density were caused by the developers having different experience and degrees of skill. However, we do not consider this possibility to be relevant for our investigation, because the JEF framework and both the DCF and S&A applications were developed within the same development unit. Around one third of the developers worked on all three of the projects. The remaining developers and testers involved in the projects have comparable skills (a Bachelor's or Master's degree in computer science) with respect to education and programming experience.

7 Conclusion and future work

Several empirical studies have compared the defect density of reused software and the software that reused them, and have observed that the reused software has lower defect densities. However, few solid studies have tried to examine the reasons for this phenomenon. We studied the defect profiles of three large industrial software systems in one company. One software system is reused by the two others as a framework. We examined all defects of these software systems (232 for the reused framework, and 592 and 793 for the other two) over all their releases. We classified the defects using ODC; compared the densities, severities, and impacts of different types of defect; and performed a follow-up qualitative RCA to find explanations for all our observations. Results of our study show that:

- The reused software has a lower defect density in total than one application that are reusing it, and has a slightly higher defect density than the other. The systematic reuse policy of the investigated company, e.g. to define and design the reused software well, keep the reused software stable, and test the reused software thoroughly, has helped to reduce the defect densities of the reused software. The relatively simple functionality and business logic of the reused software have also helped to reduce the defect density of the reused software. However, the reused software has a large amount of GUIs that are not well implemented. These GUI-type defects partly lead to a higher defect density in total of the reused software than one of the applications that are reusing it, namely S&A.
- With respect to the most severe defects, the reused software has similar defect densities to the two applications that are reusing it. However, the defect types with the highest critical defect densities of the reused software are different from those of the applications that are reusing it.

Our results deepen our understanding of the overall cause-effect relationship between software reuse and the lower defect density of the reused software. The results should induce industrial practitioners to implement more systematic reuse policies to improve the defect density of the reusable software. For researchers, the results indicate that a set of diverse decision factors have to be considered when discussing the relationship between software reuse and lower defect density.

High defect density in a pre-delivery release may be a good indicator of extensive testing, rather than of poor quality (Fenton et al. 2000). Hence, defect density cannot be used as a standard measure of quality, but defects that remain after testing will affect reliability.

Due to the internal use of the reusable and non-reusable software, our main focus was on the defects introduced by the developers (later detected either by testers or users). So, our main contribution concerns the profiles of the defects and the reasons, and not the overall quality (e.g. reliability, performance, time-to-market etc.) of the reusable and non-reusable software. A further study will be done to measure these aforementioned and other quality attributes over time for the reusable framework and applications that reuse it.

One interesting question raised by our study is how to use different Quality Assurance (QA) methodologies to improve the lower defect density of the reused software and software that reuses it. Given that reused software has different profiles of the most popular and severe defects from the software that reuses it, reused software may need to be tested in ways different from those that are used to test the applications that reuse it. A further study will investigate how to adapt the QA process of the investigated company according to the characteristics and defect profiles of the reusable software and software that reuses it.

Acknowledgements

This work was financed in part by the Norwegian Research Council for the SEVO (Software EVOlution in Component-Based Software Engineering) project (SEVO 2004) with contract number 159916/V30. We thank StatoilHydro ASA for involving us in their reuse projects and Dr. Parastoo Mohagheghi and Thor André Aresvik for valuable comments.

References

Baldassarre, MT et al. (2005) An industrial case study on reuse oriented development. In: Proceedings of the International Conference on Software Maintenance. IEEE Computer Society Press, Budapest, Hungary, pp. 283-292

Briand, Lionel C et al. (1998) Quality Assurance Technologies for the EURO Convention-Industrial Experience at Allianz Life Assurance. In: Proceedings of International Software Quality Week Europe. Communications of the ACM, Brussels, Belgium, pp. 1-23

Card, DN (1998) Learning from Our Mistakes with Defect Causal Analysis. IEEE Software 15(1): 56-63

Chillarege, R et al. (1992) Orthogonal Defect Classification - a Concept for in-Process Measurements. IEEE Transactions on Software Engineering. 18(1): 943-956

Emam, KE and Wieczorek, I (1998) The Repeatability of Code Defect Classifications. In: Proceedings of International Symposium on Software Reliability Engineering. IEEE Computer Society Press, Paderborn, Germany, pp. 322-333

Fenton, NE et al. (2000) Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Transactions on Software Engineering 26(8): 797-814

Frakes, WB et al (2001) An industrial study of reuse, quality, and productivity. Journal of Systems and Software 57(2): 99-106

Freimut (2001) Developing and Using Defect Classification Schemes. IESE-Report No. 0720.01/E. 2001

Grady, R.B (1992) Practical Software Metrics for Project Management and Process Improvement, Prentice Hall

Huber, JT (2000) A Comparison of IBM's Orthogonal Defect Classification to Hewlett Packard's Defect Origins, Types and Modes. In: Proceedings of International Conference on Applications of Software Measurement. San Jose, Ca, pp.1-17. http://www.stickyminds.com/stickyfile.asp?i=1764291&j=52901&ext=.pdf. Accessed May 2008.

IBM (2008) ODC classification. Available via IBM's website. http://www.research.ibm.com/softeng/ODC/ODC.HTM. Accessed June 2006 and January 2008.

IEEE (1994) IEEE Standard 1044-1993. IEEE Standard Classification for Software Anomalies. IEEE

Leszak. M et al. (2000) A Case Study in Root Cause Defect Analysis. In: Proceedings of the International Conference on Software Engineering. IEEE Computer Society Press, Limerick Ireland, pp. 428 – 437

Lim, WC (1994) Effect of Reuse on Quality, Productivity and Economics. IEEE Software 11(5): 23-30

Mohagheghi, P et al. (2004) An Empirical Study of Software Reuse vs. Defect Density and Stability. In: Proceedings of the International Conference on Software Engineering. IEEE Computer Society Press, Edinburgh, Scotland, pp. 282-291

Mohagheghi, P et al. (2007) Quality, Productivity and Economic Benefits of Software Reuse: A Review of Industrial Studies. Journal of Empirical Software Engineering 12(5): 471-516

Morad, S et al. (2005) Conventional and open source software reuse at Orbotech- an industrial experience. In: Proceedings of the International Conference on Software- Science, Technology & Engineering. IEEE Computer Society Press, Herzliyah, Israel, pp. 110-117

Morisio, M et al. (2002) Success and Failures in Software Reuse. IEEE Transactions on Software Engineering 28(4): 340-357

Rothenberger, MA. et al. (2003) Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices. IEEE Transactions on Software Engineering 29(9): 825-837

Selby, W (2005) Enabling Reuse-Based Software Development of Large-Scale Systems. IEEE Transactions on Software Engineering 31(6): 495-510

SEVO (2004) Software Evolution in Component-Based Software Engineering. http://www.idi.ntnu.no/grupper/su/sevo/. Accessed June 2006.

Sindre, G et al. (1995) The REBOOT Approach to Software Reuse. Journal of Systems and Software 30(3): 201–212

Succi, G et al. (2001) Analysis of the Effects of Software Reuse on Customer Satisfaction in an RPG Environment. IEEE Transactions on Software Engineering 27(5): 473-479

Thomas, WM et al. (1997) An analysis of errors in a reuse-oriented development environment. Journal of Systems and Software 38(3): 211-224

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in Software Engineering—an Introduction. Kluwer Academic Publishers

Zhang, W et al. (2005) Reuse without compromising performance: industrial experience from RPG software product line for mobile devices. In: Proceedings of the International Software Product Line Conference. IEEE Computer Society Press, Rennes, France, pp. 57-69

Zheng, J et al. (2006) On the Value of Static Analysis for Fault Detection in Software. IEEE Transactions on Software Engineering 32 (4): 240-253

Appendix A: Defect classification schemes and definitions of defect types

A defect classification scheme is used to characterize the nature of defects. There are three major schemes for classifying defects: the IEEE 1044 standard (IEEE 1994), the Hewlett-Packard (HP) Scheme (Grady 1992), and IBM's Orthogonal Defect Classification (ODC) scheme (Chillarege et al. 1992). The IEEE scheme provides too many attributes and classifications (more than 140), and in too much detail, to be used effectively in practice. The HP scheme includes attributes for defining the origin, type, and mode. The goal of the HP scheme is to initiate the improvement of processes and the early detection of defects. However, it lacks an attribute to define what the user will experience, if the defect persists after the application has been deployed at the user's site. The goal of IBM's ODC scheme is to associate each defect type with a specific stage of development. It is more suitable to use the ODC scheme than the HP scheme when the primary objective is to examine closely trends regarding defects throughout the lifecycle of the project (Huber 2000). While all ODC attributes capture the semantics of a defect (Chillarege et al. 1992), the attributes "defect type, trigger, and impact" play a crucial role in the scheme. Detailed explanations of each attribute value may be found at (IBM 2008; Emam and Wieczorek 1998). ODC has been employed to obtain a first overview of the defects. For example, Briand et al. (1998) classified the defects found in newly introduced inspections according to the impact attribute of ODC in order to characterize the defects found in terms of their visibility to the user. ODC can also be used to evaluate and improve technology. For example, in order to investigate the value of automatic static analysis, the defects found by the static analysis and those not found by this technique can be classified (Zheng et al. 2006).

Emam and Wieczorek (1998) indicate that the use of ODC is, in general, repeatable in many areas of software engineering, although there is no alignment between the Target (which represents the identity of the work product where the fix was implemented) and the type of defect (Huber 2000).

A few studies indicate that ODC can be adapted in minor ways according to project contexts (Emam and Wieczorek 1998; Freimut 2001). In our study, we ran a trial classification on the defects using ODC and found that some defects cannot be classified by classical ODC (Emam and Wieczorek 1998). Thus, we added two defect types, namely GUI-type and data-type. The definitions of the types of defect used in our study are shown in Table A.1.

Table A.1 Definition of different defect types

Defect type	**Definition**	**Examples**
Assignment /Initialization	Value(s) assigned incorrectly or not assigned at all; but note that a fix involving multiple assignment corrections may be of the type Algorithm.	1) Internal variable or variable within a control block did not have the correct value, or did not have any value at all. 2) Initialization of parameters 3) Resetting a variable's value. 4) The instance variable that captures a characteristic of an object (e.g., the colour of a car) is omitted. 5) The instance variables that capture the state of an object are not correctly initialized.
Checking	Errors caused by missing or incorrect validation of parameters or data in conditional statements. It might be expected that a consequence of checking for a value would require additional code, such as a "do while" loop or branch. If the missing or incorrect check is the critical error, checking would still be the type chosen.	1) Value greater than 100 is not valid, but the check to make sure that the value was less than 100 was missing. 2) The conditional loop should have stopped on the ninth iteration, but it kept looping while the counter was <= 10.
Algorithm/Method	Efficiency or correctness problems that affect the task and can be fixed by (re)implementing an algorithm or local data structure without the need to request a design change. Problem in the procedure, template, or overloaded function that describes a service offered by an object.	1) The low-level design called for the use of an algorithm that improves throughput over the link by delaying transmission of some messages, but the implementation transmitted all messages as soon as they arrived. The algorithm that delayed transmission was missing. 2) The algorithm for searching a chain of control blocks was corrected to use a linear-linked list instead of a circular-linked list. 3) The number and/or types of parameters of a method or an operation are specified incorrectly. 4) A method or an operation is not made public in the specification of a class.
Function/Class/Object	The error should require a formal design change, because it affects significant capability, end-user interfaces, product interfaces, interface with hardware architecture, or global data structure(s); The error occurred when implementing the state and capabilities of a real or an abstract entity.	1) A database did not include a field for street address, although the requirements specified it. 2) A database included a field for the post code, but it was too small to contain international post codes as specified in the requirements. 3) A C++ or SmallTalk class was omitted during system design.
Timing/Serialization	Necessary serialization of shared resource was missing, the wrong resource was serialized, or the	1) Serialization is missing when making updates to a shared control block. 2) A hierarchical locking scheme is in use,

	wrong serialization technique was employed.	but the defective code failed to acquire the locks in the prescribed sequence.
Interface/O-O Messages	Communication problems between: 1) modules 2) components 3) device drivers 4) objects 5) functions via 1)macros 2)call statements 3)control blocks 4)parameter lists	1) A database implements both insertion and deletion functions, but the deletion interface could not be called. 2) The interfaces specifies a pointer to a number, but the implementation is expecting a pointer to a character. 3) The OO-message incorrectly specifies the name of a service. 4) The number and/or types of parameters of the OO-message do not conform to the signature of the requested service.
Relationship	Problems related to associations among procedures, data structures, and objects. Such associations may be conditional.	1) The structure of code/data in one place assumes a certain structure of code/data in another. Without appropriate consideration of their relationship, the program will not execute or it executes incorrectly. 2) The inheritance relationship between two classes is missing or incorrectly specified. 3) The limit on the number of objects that may be instantiated from a given class is incorrect and causes the performance of the system to degrade.
GUI	Problem related to the layout of the GUI	1) Wrong size of button 2) Meaningless information in the GUI 3) Wrong text colour
Data	Structure, content, declaration	Are files opened with the right permissions? Are the correct data files accessed? Are there any missing variables for the object definition? Are variable definitions the right size to hold the data?

Appendix B: Definitions of impacts

In this study, we used the definition of impacts of the classical ODC (Emam and Wieczorek 1998), as shown in Table B.1.

Table B.1 Definition of different types of defect

Name	**Definition**	**Examples**
Installability	The ability of the client to prepare and place the software in position for use. (Does not include Usability).	1) During automated installation, received an error message saying installation failed because a file was missing.
Integrity/Security	The protection of systems, programs, and data from inadvertent or malicious destruction, alteration, or disclosure.	1) Logged in as Read Only, Profiles enabled. Was able to save changes from the System Component Assignment Panel. Was also able to delete a component.
Performance	The speed of the software as perceived by the client and the client's end users, in terms of their ability to perform their tasks.	1) Module ISGGRP00 should not hold the GRS local lock for so long that it causes the rest of the complex to hang. After processing a certain number of requests it should release and then re-obtain the lock in order to give other units of work a chance to execute.

Maintenance	The ease of applying preventive or corrective fixes to the software. An example would be that the fixes can not be applied due to a bad medium. Another example might be that the application of maintenance requires a great deal of manual effort, or is calling many pre- or co-requisite maintenance.	1) Fixes can not be applied due to a bad medium. 2) Maintenance requires a great deal of manual effort.
Serviceability	The ability to diagnose failures easily and quickly, with minimal impact on the client	1) The diagnostics software numbers error messages, rather than indicating where the problem actually occurred.
Migration	The ease of upgrading to a current release, particularly in terms of the impact on existing client data and operations. This would include planning for migration, where a lack of adequate documentation makes this task difficult. It would also apply in those situations where a new release of an existing product introduces changes that affect the external interfaces between the product and the client's applications.	1) Co-requisite information with regard to other products is not made available to clients. 2) When migrating to a new level, the client's applications fail because the external interface has been changed to no longer accept blanks. This ?lack of? backward compatibility forces the client to rewrite 36 applications.
Documentation	The degree to which the publication aids provided for understanding the structure and intended uses of the software are correct and complete.	1) MSGISG015I RCAAE78 is not documented in the system messages manual.
Usability	The degree to which the software and publication aids enable the product to be understood easily and used conveniently by its end user.	1) In some situations, the date field is not filled in. 2) When running several jobs in a system test, the system was flooded with messages. They scrolled by so quickly that they could not be read. 3) In order to perform a specific migration task, the client must enter many commands, some with parameters that contain information that it is difficult to find and understand.
Standards	The degree to which the software complies with established pertinent standards.	1) Command menu occurs on the bottom of the screen, instead of at top (which is the industry standard.) 2) Protocol specifications for participating in an exchange across heterogeneous systems are not being followed.
Reliability	The ability of the software to perform its intended function consistently without unplanned interruption. Severe interruptions, such as ABEND and WAIT would always be considered reliability.	1) While invoking modem software, the system crashed and had to be rebooted.
Capability	The ability of the software to perform its intended functions, and satisfy KNOWN requirements, where the client is not affected in any of the previous categories.	1) On an unconditional Latch Obtain request for an SRB, the code in ISGLRTR does not check the return code from SUSPEND SPTOKEN. If there is a user or system error, this could result in the requester thinking that the latch had been obtained when in fact, it has not. 2) When SAVE was clicked on, nothing happened.

P6: Change Profiles of a Reused Class Framework vs. two of its Applications

Submitted to Journal of Information and Software Technology, 01.07.2008.

Anita Gupta[1], Jingyue Li[1], Reidar Conradi[1], Harald Rønneberg[2] and Einar Landre[2]

[1] Dep. of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU), Trondheim, Norway
{anitaash, jingyue, conradi}@idi.ntnu.no

[2] StatoilHydro ASA KTJ/IT, Forus, Stavanger
{haro, einla}@statoilhydro.com

Abstract Software reuse is expected to improve software productivity and quality. Although many empirical studies have investigated the benefits and challenges of software reuse from development viewpoints, few studies have explored reuse from the perspective of maintenance. This paper reports on a case study that compares software changes during the maintenance and evolution phases of a reused Java class framework with two applications that are reusing the framework. The results reveal the following. 1) The reused framework is more stable, in terms of change density, than one application that is reusing it and more unstable than the other. 2) The reused framework has profiles for change types that are similar to those of the applications, where perfective changes dominate. 3) The lifecycle of both the reused framework and its applications is the same: initial development, followed by a stage with extending capabilities and functionality to meet user needs, followed by a stage in which only minor defect repairs are made, and finally, phase-out. However, the reused framework goes faster from the stage of extending capabilities to the stage in which only minor defect repairs are made than its applications. 4) The factors that affect the change densities and change profiles of both framework and applications are functionalities, development practice, complexity, size, and age. Thus, all these factors must be considered to predict change profiles in the maintenance and evolution phase of software.

Keywords: *Software reuse, Software change, Case study, Class framework*

1. Introduction

After a software system is delivered to the client for use, it will evolve and be maintained continuously until it is replaced or discarded [Bennett00]. Pigoski [Pigoski97] found that the percentage of the IT industry's expenditure on maintenance was 40% in the early 1970s, 55% in the early 1980s, and 90% in the early 1990s. Krogstie et al. [Krogstie06] conducted a survey that investigated the development and maintenance of business software in Norway. The same survey was performed in 1993 and 1998. The results show that overall, about 40% of available time is spent on maintenance. Software reuse is expected to utilize past accomplishments, to facilitate software development productivity, and to improve the quality of the developed software system [Lim94]. Several studies have investigated whether software reuse and component-based development have the potential to facilitate the maintenance and evolution of a system. They concluded that reused components are more stable than non-reusable ones [Frakes01][Algestam02][Mohagheghi04a][Selby05][Zhang05]. However, one study concludes the opposite, stating that reused components with major revisions later will need more changes per source line than newly developed components [Selby05].

To investigate the benefits and challenges of software reuse with respect to maintenance, we performed an industrial case study to compare the actual changes performed on a reusable framework called the Java Enterprise Framework (JEF). This framework is developed by the IT-department of a large Norwegian Oil & Gas company. It is reused "as-is" by two applications, Digital Cargo Files (DCF) and Shipment and Allocation (S&A) in the same company. We formulated two research questions:

RQ1: *Whether the reused software experiences fewer or more changes than its applications, and the likely reasons for the differences or similarities.*

RQ2: *Whether the reused software experiences the same profile of changes over time with the software reusing it, and the reasons for the differences and similarities.*

We selected files from the three systems at random, classified all changes on these files, and compared the distribution of different types of changes (perfective, corrective, adaptive and, preventive) in general and over time. In addition, we performed a root cause analysis (RCA) to interpret the results. The novelties of the study are:

- *It compared the maintenance activities of reused software vs. non-reused software by classifying changes into perfective, corrective, adaptive, and preventive types of change.*
- *It compared the change profile of reused and non-reused software over time.*
- *It used root cause analysis to investigate the reasons for software evolution and maintenance activities.*

This study contributes to the understanding of software reuse and software changes, and concludes that:

- *Software reuse does not necessarily lead to stable software. In our case, the reused framework has a lower change density than one application reusing it and a higher change density than the other application.*
- *Perfective changes (i.e. caused by new or changed requirements, as well as optimizations) constitute the highest percentage of changes in both the reused framework and in the applications reusing it, followed by corrective changes.*
- *Both the reused framework and the applications experienced the following lifecycle: initial development, then extending the capabilities and functionalities of the system to meet user needs, followed by repairing minor defects. However, the reused framework experienced only one such lifecycle, while the applications experienced several.*
- *Kemerer [Kemerer97] concludes that five main factors (i.e. software functionality, software complexity, development practices, software size, and software age) will affect possible maintenance activities. Regarding the change densities and change profiles of both the framework and the applications, those of Kemerer's factors [Kemerer97] that affect the maintenance activity most in our case are functionality and development practices, followed closely by complexity. The factors that affect it least in our case are age and size. The functionalities and development practices of the software usually influence the future change density and the type of change (perfective, corrective, adaptive, and preventive).*

The remainder of the paper is structured as follows. Section 2 presents related work. Section 3 presents the motivation for the research and research questions. Section 4 describes the research design. Section 5 presents the results. Section 6 discusses these results Section 7 concludes.

2. Related work

Understanding the issues related to changes involved in the evolution and maintenance of software has been a focus of software engineering study since the 1970s. The goal has been to identify the origin of changes, as well as their type, relative frequency of occurrence, and effort required to make them. Software changes are important because they account for a major part of software costs. At the same time, they are necessary; the ability to alter software quickly and reliably means that businesses can take advantage of new opportunities and can remain competitive [Bennett00]. Lehman [Lehman80] carried out the first empirical studies of software changes, finding that systems that operate in the real world have to be adapted continuously; otherwise, their usability and relevance decreases rapidly. Software systems usually need to be changed during their lifetime because the original requirements may change to reflect changing business, user, and client needs [Postema01].

2.1. Studies on the distributions of different types of software changes

One kind of study that has been performed on software changes examines the *static* aspect of changes, i.e. the distribution of different kinds of change, or the distribution of effort spent on performing different kinds of change. Table 1 summarizes the nature and conclusions of these studies.

Table 1. Studies related to distributions of different changes

Study description	Distribution and definition of different types of change	Other observations of the study
A questionnaire-based survey that collected data from 69 systems, which were developed using different programming languages, e.g. Cobol, Assembler, Fortran etc. [Lientz78].	- 60% **perfective** (enhancements and speed performance) - 18% **adaptive** (changes to data inputs and files) - 17% **corrective** (emergency fixes and debugging) - 4% **other** (no description given)	User demands for enhancements and extensions constitute the most important problem area with respect to management.
A case study that investigated change requests collected for two years in a Canadian financial institute [Abran91]. Used the same definitions as [Lientz78] for corrective, adaptive, and perfective changes. Analyzed 2152 change requests.	- 60% **adaptive** - 21% **corrective** - 3% **perfective** - 15% **user support** (handle user requests of application rules and behaviour, requests for work estimates, requests for preliminary analysis)	Maintenance team in 1989 spent 64% of their time doing maintenance work (e.g. optimization and adding new functionality) other than correcting defects and errors.
A survey conducted in the MIS (Management Information System) department in nine different application domains in Hong Kong. 1000 questionnaires were sent out and about 50 responses were received [Yip94].	- **perfective** (40% enhancements, 7% tuning, and 6% reengineering) - 16% **corrective** (correct faults) - 10% **adaptive** (adaptation to new environment) - **other** (13% answering questions and 9% documentation)	In Hong Kong, 66% of the total software life cycle effort was spent on software maintenance. The most cited maintenance problems were staff turnover, poor documentation, and changing user requirements.
A structured interview with managers and maintainers in a computer department of a large Norwegian telecom organisation in 1990-1991 (study1) and 1992-1993 (study2) [Jørgensen95]. Systems were developed using either Cobol or Fourth Generation languages.	**Results of interviews with managers:** - 44% **perfective** (changes in user requirements) - 29% **adaptive** (make software usable in a changed environment) - 19% **corrective** (correct faults) - 8% **preventive** (preventing problems before they occur) **Results of interviews with maintainers:** - 45% **perfective** - 40% **adaptive** - 9% **corrective** - 6% **preventive**	If the amount of corrective work is calculated on the basis of interviews solely with managers, it will be as twice as much as the actual work reported in logs (i.e. the amount of corrective work may be exaggerated in interviews).

Studied 10 projects conducted in the Flight Dynamic Division (FDD) in NASA's Goddard Space Flight Center. The FDD maintains over 100 software systems totalling about 4.5 million lines of code. 85% of the systems are written in FORTRAN, 10% in Ada, and 5% in other languages [Basili96].	- 61% **perfective** (improve system attributes and add new functionality) - 20% **other** (e.g. management, meeting etc.) - 14% **corrective** (correct faults) - 5% **adaptive** (adapt system to new environment) -	Error corrections are small isolated changes, while enhancements are larger changes to the functionality of the system. More effort is spent on isolation activities in correcting code than when enhancing it.
A case study investigated the change of maintenance requests during the lifecycle of a large software application (written in SQL) [Burch97]. Analyzed 654 change and maintenance requests.	- 49% **repair** (fixing bugs) - 26% **enhancement** (add or modify functionalities) - 25% **user support** (consulting and answering user requests)	User support reaches its peak in the 4th month (first stage). Repair reaches its peak in the 13th and 14th months (second stage), while enhancement is the dominant factor in the third stage (25th month).
A survey carried out in financial organizations in Portugal. Data was collected from 20 project managers [Sousa98].	- 49% **adaptive** (changes in platform) - 36% **corrective** (error modifications) - 14% **perfective (**expand system requirements and optimization) - 2% **preventive** (future maintenance action)	3% of the respondents considered the software maintenance process to be very efficient, while 70% considered that efficiency is very low.
An Ada system of the NASA Goddard Space Flight Center [Evanco99]. Analyzed 453 non-defect changes.	- 31% **planned enhancements** (anticipated at the start of development) - 30% **other** (code debugging, enhancements and maintainability) - 29% **requirements modifications** (implementation of requirement changes) - 10% **optimization** (optimize software performance)	Changes related to optimizations require the most effort to isolate, while planned enhancements require the most effort to implement.
A subsystem that contains 2 million lines of source code [Mockus00]. Analyzed 33171 modification requests.	- 46% **corrective** (fixing faults) - 45% **adaptive** (adding new features) - 5% **inspection** (code checking to figure errors) - 4% **perfective** (code restructuring)	Corrective changes tend to be the most difficult, while adaptive changes are difficult only if they are large. Inspection changes are perceived as the easiest.
A case study on re-engineering a people-tracking subsystem of an automated surveillance	- 38% **perfective** (optimization, restructuring and adding new functionalities)	The effort required to adapt the system was high, because the software needed to be

system, which was written in C++ and had 41 KLOC [Satpathy02]. Analyzed the distribution of maintenance effort during the whole maintenance phase.	- 31% **adaptive** (adapting to changed environments) - 23% **preventive** (preventing malfunctions and improving maintainability) - 8% **corrective** (correcting problems)	ported to a different platform.
Examined three software products: – A real-time product written in a combination of assembly language and C. Data of 138 modified versions were collected. – The Linux kernel. Data from 60 modified versions were collected. – GCC (GNU Compiler Collection). Data from 15 versions were collected. [Schach03].	The analysis and collection of data were performed at two levels, using the same definition as [Lientz78]: (1) **change log level**, i.e. each entry in the change log was regarded as one unit of change. (2) **module level**, i.e. all the changes made to a module were regarded as a single unit of maintenance. **Change log level:** - 57% **corrective** - 39% **perfective** - 2.4% **other** - 2.2% **adaptive** **Code module level:** - 53% **corrective** - 36% **perfective** - 4% **adaptive** - 0% **other**	All three maintenance categories were statistically very highly significantly different from the results of [Litentz78]. Corrective maintenance was more than three times the level of the results of [Litentz78].
Four releases of a telecommunication system written in Erlang, C, Java, and Perl. [Mohagheghi04b]. Analyzed 187 change requests.	- 61% **perfective** (new or changed requirements as well as optimization) - 19% **adaptive** (adapting to new platforms or environments) - 16% **preventive** (restructuring and reengineering) - 4% **other** (saving money/effort) Corrective changes are reported elsewhere.	There is no significant difference between reused and non-reused components in the number of change requests per KSLOC.
Web-based Java application, consisting of 239 classes and 127 JSP files [Lee05]. Based on Swanson's definition [Swanson76] and Kitchenham's ontology [Kitchenham99]. Analyzed 93 fault reports	**Based on Swanson's definitions:** - 62% **perfective** - 32% **corrective** - 6% **adaptive** **Based on Kitchenham's ontology:** - 68% **enhanced maintenance** - 32% **corrective**	Maintenance effort of Java application is similar to the distribution in previous non object-oriented and non web-based applications.

A close investigation of studies in Table 1 reveals that:

- Different studies classify changes differently, noticed by [Chapin01].
    - o Four studies classified changes into four categories: adaptive, corrective, perfective, and preventive [Jørgensen95][Sousa98][Satpathy02][Mohagheghi04b][Lee05].

    - Several studies did not include preventive changes and classified the changes into adaptive, corrective, and perfective, with a fourth category of user support in [Abran91], inspection in [Mockus00], and "other" in [Lientz78][Yip94][Basili96][Schach03].
    - One study classified changes into planned enhancement, requirement modifications, optimization, and "other" [Evanco99].
    - One study classified changes into user support, repair, and enhancement [Burch97].
- Definitions of different types of change are slightly different. For example, perfective change is defined as user enhancements, improved documentation, and recoding for computational efficiency in [Lientz78], and as restructuring the code to accommodate future changes in [Mockus00]. It is also defined as encompassing new or changed requirements (expanded system requirements) as well as optimization in [Sousa98][Mohagheghi04b], and is defined as enhancements, tuning, and reengineering in [Yip94].
- The distributions of different types of change are not the same for different systems. 62% of studies, including [Lientz78][Yip94][Basili96], found that perfective changes (the median value of perfective changes of those studies presented in Table 1 is 57%) were the most frequent. However, perfective changes in the system in [Mockus00] were the least frequent. Twenty-three percent of the studies, reported by [Burch97][Mockus00], found that corrective changes were the most frequent. Fifteen percent of the studies, including [Abran91][Sousa98], found that adaptive changes were the most frequent.

2.2. Studies on software changes over time

Another kind of study on software changes investigated how the changes vary over time (the *longitudinal* aspect.) Gefen and Schneberger [Gefen96] examined an information system for 29 months and reported that in the first stage, the software was stabilized within the framework of its original specifications and changes were centered on corrective modifications. In the second period, the software was improved and new functions were added to the original framework. In the third period, the system was expanded beyond its original specifications by adding many new applications. Burch and Kung [Burch97] studied a large application for 67 months and reported that user support reaches its peak in the first stage. Repair is prevalent in the second stage and enhancement is the dominant factor in the third stage. Rajlich and Bennett [Bennett00] proposed a stage model to describe the lifecycle of a software system, as shown in Figure 1. According to that model, the software life cycle consists of five distinct stages:

- *Initial development.* Engineers develop the system's first functioning version.
- *Evolution.* Engineers extend the capabilities and functionality of the system to meet user needs, possibly in major ways.
- *Servicing.* Engineers repair minor defects and make simple functional changes.
- *Phase-out.* The company decides not to undertake any more servicing, seeking to generate revenues from the existing system for as long as possible.
- *Close-down.* The company withdraws the system.

A variation of the stage model is the versioned staged model [Bennett00], also shown in Fig. 1. The backbone of the versioned staged model is the evolution stage. At certain intervals, a company completes a version of its software and releases it to clients. Evolution continues, with the company eventually releasing another version and only servicing the previous version.

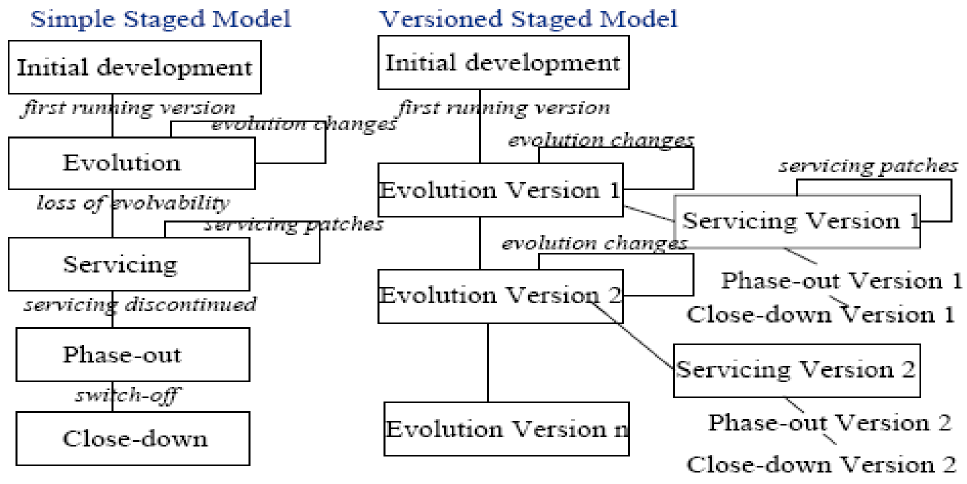

Fig. 1. Simple staged model vs. Versioned staged model [Bennett00].

2.3 Studies on software changes and software reuse

A few studies [Frakes01][Algestam02][Mohagheghi04a][Selby05] have examined the possible influence of software reuse on the changes of a system, as shown in Table 2.

Table 2. Studies comparing changes of reusable components vs. those in non-reusable ones

Quality focus	**Quality measures**	**Conclusion**
Change density	Number of change requests per source code line.	Reused components are more stable in terms of volume of code modified between releases [Mohagheghi04a].
	Percentage of code-line changes (enhancement or repair).	The modules reused with major revision (>=25% revision) had the most code changes per SLOC [Selby05].
Number of changes	The number of changes (enhancement or repair) to a module	More reuse results in fewer changes [Frakes01].
Amount of modified code	Size of modified or new/deleted code/total size of code per component between releases.	Non-reused components are modified more than reused ones [Mohagheghi04a].
Number of change scenarios	Number of changes to which a software system is exposed (e.g. adding communication protocols, porting to new platforms, issues related to the database manager, etc.)	Reusing components and a framework resulted in increased maintainability in terms of cost of implementing change scenarios [Algestam02].

Although most studies [Frakes01][Algestam02][Mohagheghi04a] in Table 2 conclude that *software reuse is significantly correlated to fewer changes or lower defect density*, one study observed that a reused module that undergoes major revision has the most changes per source line [Selby05]. A close investigation of the studies in Table 2 further illustrates that:

- None of the studies performed detailed analyses (as was the case with studies listed in Table 1). "Detailed analysis" here refers to dividing the changes into different types and comparing the distribution of the changes according to type. Several factors, e.g. complexity, functionality, development practice, age, and size may determine the profile of software maintenance [Kemerer97]. Thus, *comparing only the number or density of the defects is not sufficient to warrant the conclusion that software reuse is significantly* *correlated to fewer changes*. Thus, further study is needed to investigate the relation between software reuse and software changes of different types.

3. Research motivation and research questions

No study listed in Table 2 performed analysis similar to that in [Gefen96][Burch97], i.e. compared the changes of reused software and the software reusing it over time. Thus, it remains an open question whether reused software and its applications follow similar or different change profiles or software lifecycle [Bennett00] over time. Answers to this question would make it easier for software maintainers to plan maintenance effort according to possible change profiles of reused software and software reusing it.

To determine whether *software reuse actually leads to fewer software changes,* we decided to compare the software changes that are made to reusable vs. non-reusable software. We investigated two research questions (see Section 3.1) to address the limitations of the studies in Table 2.

3.1. Research questions

Research question **RQ1** is: ***Whether the reused software experiences fewer or more changes than its applications, and the likely reasons for the differences or similarities.*** To examine this research question in detail, we designed three subquestions as follows:

RQ1.1. What are the total change densities of reused software and of non-reused software?
RQ1.2. What are the densities of the individual change types of reused software and of non-reused software?
RQ1.3. What are the reasons for the answers to research questions RQ1.1 and RQ1.2?

The research questions **RQ2** is formulated as: ***Whether the reused software experiences the same profile of changes over time with the software reusing it, and the reasons for the differences and similarities.*** To examine this research question in detail, we designed two subquestions RQ2.1 and RQ2.2:

RQ2.1. What are possible differences of change profile of software being reused and software reusing it?
RQ2.2. What are the reasons for the answers to RQ2.1?

4. Research design

To answer our research questions, we investigated three software systems from the largest Norwegian oil and gas company, StatoilHydro ASA. In this section, we first introduce the company, the three systems, and software change data of these systems. After that, we describe how the change data were analyzed and how the follow-up RCAs [Card98] were performed.

4.2. Data collection

4.2.1 The investigated company

StatoilHydro ASA has its headquarters in Norway and has branches in 40 countries. It has 31,000 employees in total. The IT department of the company is responsible for developing and delivering domain-specific software to the host organization, so that key business areas can become more flexible and efficient in their standard operations. It is also responsible for the operation and support of IT systems. This department consists of approximately 100 developers, who are located mainly in Norway. In addition, the company subcontracts many software development and operations to consulting (software) companies. These subcontracting operations may involve over 1000 ICT specialists.

4.2.2 The software systems investigated

The company initiated its reuse strategy in 2003 with pre-studies. Then, a reusable software framework was developed. This framework (Java class library) is based on J2EE (Java 2 Enterprise Edition), and is a Java class framework for developing Enterprise Applications. Thus, the framework is called the “JEF framework” (hereafter, JEF). The JEF consists of seven separate components or modules (i.e. each component or modules consist of various library classes) that can be reused separately or together.

JEF release 1 was finished around June 2005. PDM (Physical Deal Maintenance) was the first application to use it, in the summer of 2005. In this period, some weaknesses in the framework were discovered. Changes that were made in response to these weaknesses were incorporated into JEF release 2 in September 2005. The DCF application reused JEF release 2 during late summer and autumn of 2005. After DCF reused the JEF, further minor changes were made to the framework. These changes were finished by early November 2005, when JEF 3 was released. The second application, S&A, was developed during early 2006 and reused JEF release 3.

The relations between the JEF and applications using/reusing it are shown in Fig. 2. The size (measured in non-commented source lines of code (NSLOC)) and release date of the reusable vs. non-reusable software systems are shown in Table 3. Detailed information on the JEF, DCF and S&A are presented in Appendix A. The company has the same test team and the same test coverage for both reusable and non-reusable software. For instance, for unit testing, 85% of the code lines are executed by unit tests to make sure the code works as expected. We did not include defects in the PDM application other than those in the JEF in our study, because PDM was the first application to *use* JEF; it did not *reuse* it (like DCF and S&A). Hence, defects from PDM were not analyzed.

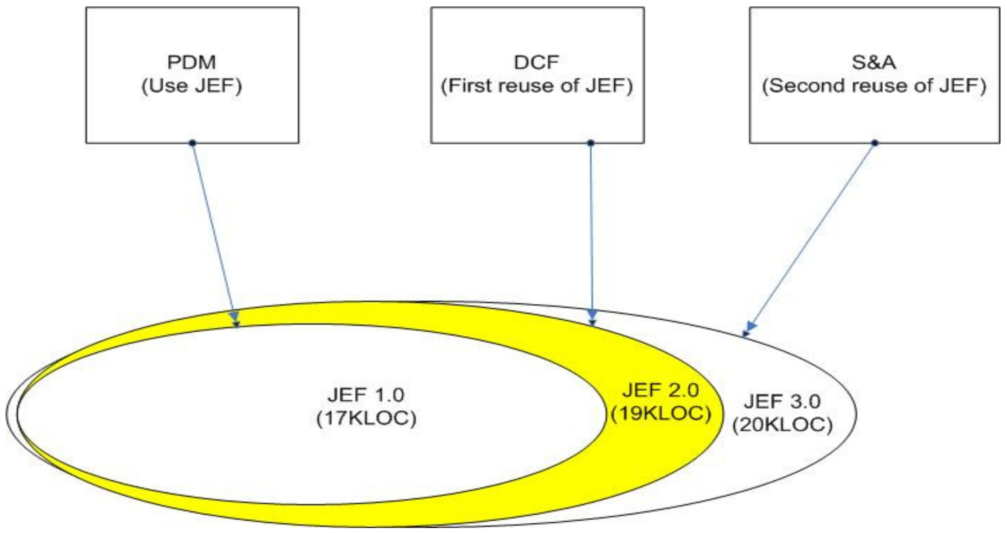

Fig.2. The relation between the JEF, DCF, and S&A.

Table 3. Size and release date of the three systems

System	Release1		Release 2		Release 3	
	Date	Size (NSLOC)	Date	Size (NSLOC)	Date	Size (NSLOC)
JEF	14. June 2005	16 875	9. Sept. 2005	18 599	8. Nov. 2005	20 348
DCF	1. Aug. 2005	20 702	14. Nov. 2005	21 459	8. May 2006	25 079
S&A	2. May 2006	29 957	6. Feb. 2007	50 879	12. Dec. 2007	64 319

From Table 3, we can see that the framework and the applications are growing. The JEF consists of seven components. These were used in PDM and reused in DCF and S&A. However, DCF and S&A were not used in any other applications. The JEF is reused in DCF and S&A and in other projects "as-is". This is how we can say that the JEF is reusable, while DCF and S&A are non-reusable. The JEF, DCF, and S&A will grow because when clients use the applications they will make some changes to them, which will also require changes to the JEF. For instance, adding new functionality to the reusable and non-reusable software will result in growth for the JEF, DCF, and S&A. Another explanation of the growth of the framework and the applications is that when a defect is found in one release (e.g. JEF 1.0), the fixes will be included in the next release (e.g. JEF 2.0), and so on.

4.2.3. Collected software change data

To handle changes in requirements or implemented artifacts, Change Requests (CRs) are written (by test manager or developers) and stored in the Rational ClearQuest tool. Examples of change requests are to add, modify or delete functionality; solve a problem with major design impact; or adapt to changes from, for example, JEF component interfaces. CCB stands for the *Change Control Board* (usually found in SCM systems). The project leader in StatoilHydro ASA constitutes the CCB in this context. The CCB is responsible for approving or rejecting a CR, and distributing the approved change requests (from Rational ClearQuest) among the developers. After the approved CRs have been distributed, the developers access the source files in the version control system, i.e. Rational ClearCase, to make the necessary changes. When implementing the changes, the developers follow the following steps:

- They check-out the files that correspond to the CR that they are working on.
- They implement changes on the checked-out files, possibly locking the branch that they are working on.
- They give the file a change description, which is a thorough description that elaborates on what changes they have made, and a time and date (timestamp).
- Finally, they check the changed files back into Rational ClearCase.

Due to the fact that Rational ClearCase captures information about all source code and other software changes, the first author of the paper extracted the following data for each file (of JEF, DCF, and S&A) from Rational ClearCase: an ID, a filename with all its version numbers and the corresponding check-in timestamp (includes only the date), change description (prose description of the change), size for its base and last version, and location of the changes (i.e. the name of the components on which the changes were implemented). The information was extracted manually.

Over all releases, there were, in total, 481 files for the JEF, 1365 for DCF, and 405 for S&A. We selected only a subset of the files because it was too time-consuming to analyse all changes to all 2251 files. We attempted to select for analysis a representative subset of the files in each component. When we collected the data, we decided to have a threshold value of 10 files. If a component had less than 10 files, we included all the files. If there were more than 10 files, we picked a random subset that, we assumed, would be representative of the properties of the entire component. A sampling calculator [MaCorr08] was used to calculate a sufficient sample size (confidence level was 95% and confidence interval was 3%). As an example of our procedure, the component JEFClient had 195 files. Using the calculated sample size from the sampling calculator, which was 165 files, we randomly (using a mathematical function in MS Excel) selected 165 files from the JEFClient to include in the dataset. We collected data from 442 files for the JEF, 932 for DCF, and 343 for S&A.

4.3. Data analysis

To answer **RQ1.1**, we divided the total number of changes during the maintenance end evolution phases in the JEF by the total NSLOC of the collected files at the end. The same went for DCF and S&A. Given that we collected all changes that were made, from the development phase to the later maintenance phases, we divided the changes according to the date of the first release for the JEF, DCF, and S&A (see Table 3). All changes made after this date were regarded as maintenance and evolution changes. The changes made before or on this date were treated as development changes. We included only changes that were made during the maintenance and evolution phases for analysis, although the numbers of change made during the development phases are also presented.

To calculate the change density (in the source code) of different changes and to answer **RQ1.2**, we first classified the changes into different categories, according to the classification system proposed by [Kitchenham99] and used by [Lee05]. We then divided the changes in the perfective, corrective, adaptive, and preventive categories by the total NSLOC of the corresponding system (the JEF, DCF and S&A). The classification proposed in [Kitchenham99] is based on the actual maintenance activity performed. Given that each change description in the Rational ClearCase only describes what activities have been done to complete a software change, the activity-based classification of changes that is proposed in [Kitchenham99] was more suitable for our analysis and was therefore used. However, adaptive change is defined in [Kitchenham99] as "enhancements that add new system requirements". This definition does not specify whether these enhancements are related to environment or platform changes. We therefore decided to use the definition of adaptive change given in [Sommerville04]. The definitions of change categories used in our study are also similar to those in [Fenton96, pp. 354-355], and are as follows:

- *Perfective*: encompass new or changed requirements, as well as optimizations.
- *Preventive*: changes related to restructuring and reengineering.
- *Corrective*: bug fixing.
- *Adaptive*: changes related to adapting to new platform or environments.

The first and second authors of the paper classified all the change descriptions separately. They then compared the results jointly. This resulted in 100% agreement. During the classification and comparison, we noticed that some of the change descriptions were labelled as "no changes" (meaning no changes were made to the code). We grouped "no changes" into the category "other". Some other changes were labelled as "initial review". (This means that changes were performed after a formal review of the code. However, the actual changes that were performed were not described in detail). We classified "initial review" into the category "inspection", because changes in this category could not be classified exactly. After we had classified the changes, we calculated the change distribution of each type of change to the JEF, DCF and S&A.

To answer **RQ1.3**, we did a RCA [Card98] by showing all results of the **RQ1.1** to **RQ1.2** to a senior developer, who has followed all development and maintenance phases, and asking him to give explanations. To avoid possible threats to validity, this developer was not informed of our research questions. We asked him to explain the results of **RQ1.1** to **RQ1.2** from the perspectives of *functionality*, *development practice*, *software complexity*, *age*, and *size*, Because it has been claimed that these factors are determining factors of software maintenance [Kemerer97]. Kemerer and Slaughter chose these five factors on the basis of data from the literature (i.e. [Chapin85]) and from case studies. Their analysis used a multivariate regression to determine the association between these five factors and the different maintenance types (e.g. enhancements, repairs etc.), and they found these factors to be significant for software maintenance. To answer **RQ2.1**, we made a bar chart to show the distribution of different types of change over time (precise to the exact date). To answer **RQ2.2**, we did the same RCA as for **RQ1.3**.

5. Results and interpretations of the results

The numbers (and percentage) of different types of change in the investigated systems are shown in Tables 4, 5, and 6. The follow-up RCA provided explanations for the results for RQ1.1 and RQ1.2, using the five factors by [Kemerer97], as shown in Table 7.

We first compared the change densities of the three systems in **RQ1.1.** The results show that the JEF has a much lower change density (398/20348 = 19.6 per Kilo NSLOC) than DCF (2771/25079 = 110 per KNSLOC), but a much higher change density than S&A (589/64319 = 9.2 per KNSLOC) for the maintenance and evolution phases. We then compared the change densities of different change types of the three systems to answer **RQ1.2**. The results are:

- The *perfective* changes cover the highest percentage in the JEF, DCF, and S&A.
- Both S&A and DCF have higher percentages of *corrective* changes than the JEF.
- Both S&A and DCF have a higher percentage of *preventive* changes than the JEF.
- S&A has a higher percentage of *adaptive* changes than the JEF and DCF.

Table 4. JEF releases and the number of changes

JEF	Development	Maintenance and evolution			
		Before 2nd release	Before 3rd release	After 3rd release	In total
Perfective	489 (69%)	115 (39%)	46 (58%)	18 (67%)	179 (45%)
Corrective	127 (18%)	62 (21%)	8 (10%)	8 (30%)	78 (20%)
Preventive	6 (1%)	1 (0.3%)	24 (30%)	0 (0%)	25 (6%)
Adaptive	41 (6%)	52 (18%)	1 (1%)	1 (4%)	54 (14%)
Inspection	33 (5%)	62 (21%)	0 (0%)	0 (0%)	62 (16%)
Other	11 (2%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Total:	**707**	**292**	**79**	**27**	**398**

Table 5. DCF releases and the number of changes

DCF	Development	Maintenance and evolution			
		Before 2nd release	Before 3rd release	After 3rd release	In total
Perfective	1161 (62%)	910 (56%)	580 (53%)	25 (46%)	1515 (55%)
Corrective	425 (23%)	421 (26%)	282 (26%)	24 (44%)	727 (26%)
Preventive	258 (14%)	273 (17%)	213 (20%)	1 (2%)	487 (18%)
Adaptive	21 (1%)	24 (2%)	13 (1%)	0 (0%)	37 (0.1%)
Inspection	0 (0%)	0 (0%)	0(0%)	0 (0%)	0 (0%)
Other	12 (0.6%)	0 (0%)	0(0%)	5 (9%)	5 (0%)
Total:	**1877**	**1628**	**1088**	**55**	**2771**

Table 6. S&A releases and the number of changes

S&A	Development	Maintenance and evolution			
		Before 2nd release	Before 3rd release	After 3rd release	In total
Perfective	42 (43%)	3 (27%)	100 (42%)	120 (35%)	223 (38%)
Corrective	16 (17%)	8 (73%)	74 (31%)	68 (20%)	150 (26 %)
Preventive	17 (18)	0 (0%)	19 (8%)	36 (11%)	55 (9%)
Adaptive	22 (23%)	0 (0%)	41 (17%)	116 (34%)	157 (27%)
Inspection	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Other	0 (0%)	0 (0%)	2 (1%)	2 (1%)	4 (1%)
Total:	**97**	**11**	**236**	**342**	**589**

Table 7. Results of root cause analysis for RQ1.1 – RQ1.2
C: Complexity, F: Functionality, D: Development practice, A: Age, S: Size

Observation	Proposed factors					Explanations
	C	F	D	A	S	
The JEF has lower change density in total than DCF and higher change density than S&A	X	X	X			DCF had incomplete and poorly documented design, and higher time-to-market pressure (which required a large number of improvements over time), than JEF. DCF has tighter coupling and more business logic than the reusable JEF framework. S&A had a well-defined requirement and design document. However, after the JEF's initial development, it was reused in unexpected contexts, which led to many new requirements. The JEF is developed for reuse. However, it is difficult *to know in advance* all the functionality that a reusable framework may need.
The *perfective* changes constitute the highest percentage in the JEF, DCF and S&A		X	X			When the JEF was used by PDM, it was revealed that the Graphic User Interface (GUI) functionality did not satisfy client requirements. Thus, a lot of perfective changes had to be made. DCF had time-to-market pressure, unclear requirements, and incomplete design at the early phases of implementation. This led too many perfective changes later. S&A was first developed without involving the users. When the users got to see the application, many changes were made to requirements.
Both DCF and S&A have higher percentages of *corrective* changes than the JEF			X			For DCF and S&A, the developers did not have a detailed design at the beginning and a lot of corrective changes were made to functionality and design during the implementation and testing period.
Both S&A and DCF have higher percentages of *preventive* changes than the JEF		X	X			JEF did not have high time-to-market pressure during development. That resulted in a good design and less need for refactoring. Time pressure and incomplete design of DCF led to some refactoring during implementation. S&A had more rules and business logic than the JEF and DCF, which led to some refactoring.
S&A has more *adaptive* changes than the JEF and DCF		X				Compared to DCF and JEF, S&A had implemented heavier algorithms to do lift and cargo calculations efficiently and properly. So, the business logic in S&A must be adjusted to the different environments within which the application is going to be used.

To investigate **RQ1.1** and **RQ1.2** further, we also investigated the individual change profiles of the JEF, DCF, and S&A, according to Kemerer's five factors [Kemerer97]. We also identified the component that covers the highest percentage of changes and the component with the highest change density. The observations and their interpretations are presented in Tables 8, 9, and 10.

Table 8. The results of root cause analysis for change profiles of the JEF
C: Complexity, F: Functionality, D: Development practice, A: Age, S: Size

Observation	**Proposed factors**					**Explanations**
	C	F	D	A	S	
The total number of changes declined dramatically from release 1 to release 2 and 3.		X	X	X		Release 1 took two years to develop. Releases 2 and 3 took only two months each to develop. The JEF is a reusable framework and is used by several applications. New features are not incorporated into the framework unless they will be used by at least by two different projects. The company is very careful when introducing changes to this framework, because changes to the API may affect many applications.
Thirty percent of the changes made after the 2nd release and before the 3rd release were preventive.			X			During the implementation of the second release, a major refactoring was done to remove a cyclic dependency between security and session management components.
After the third release, 30% of the changes were corrective, more than the percentages in Releases 1 and 2.			X			A lot of the code in Release 2 was removed and replaced with third party components. The increase in corrective changes was due to defects caused by these replacements.
Most changes are in the JEFClient (29%) and JEFWorkBench (23%) components. Most changes in these components are perfective.		X			X	JEFClient constitutes the majority of the code. JEFClient and JEFWorkBench have GUI-related code. A lot changes were performed related to the GUI layout.
The component with the highest change density is JEFIntegration.	X	X				JEFIntegration possess the most complex code. When the JEF framework was reused by different applications (PDM, DCF etc.), new requirements for JEFIntegration emerged. The JEF was developed as a common framework to support GUI development, without knowing all the functionality that a framework may need at the early stage of the project.

Table 9. The results of root cause analysis for change profiles of DCF
C: Complexity, F: Functionality, D: Development practice, A: Age, S: Size

Observation	Proposed factors					Explanations
	C	F	D	A	S	
Most changes are in DCFClient (63%) and DCFCommon (41%) components. Most changes in these components are perfective.	X					These two components are the most complex components in DCF with respect to functionality.
The component with the highest change density is DCFEJB.		X				This component has several configuration files and contains script and XML. These files need to be updated often, according to the context of deployment.

Table 10. The results of root cause analysis for change profiles of S&A
C: Complexity, F: Functionality, D: Development practice, A: Age, S: Size

Observation	Proposed factors					Explanations
	C	F	D	A	S	
After the 3rd release, there are more adaptive changes than in Releases 1 and 2.			X			S&A is reusing the 3rd release of the JEF. Changes in the JEF affected S&A.
Most changes are in SACommon (51%) and SAClient (48%) components. Most changes are perfective.	X					These two components are the most complex components in S&A with respect to functionality.
The component with the highest change density is SAClient.	X					SAClient is one of the most complex components in S&A with respect to functionality.

5.2. Software change profile over time

To investigate RQ2.1, we analyzed the change profile of different types of change of the three systems over time, as shown in Figs. 3, 4, and 5. The RCAs of the observed change profiles are presented in Table 11, according to Kemerer's five factors from [Kemerer97].

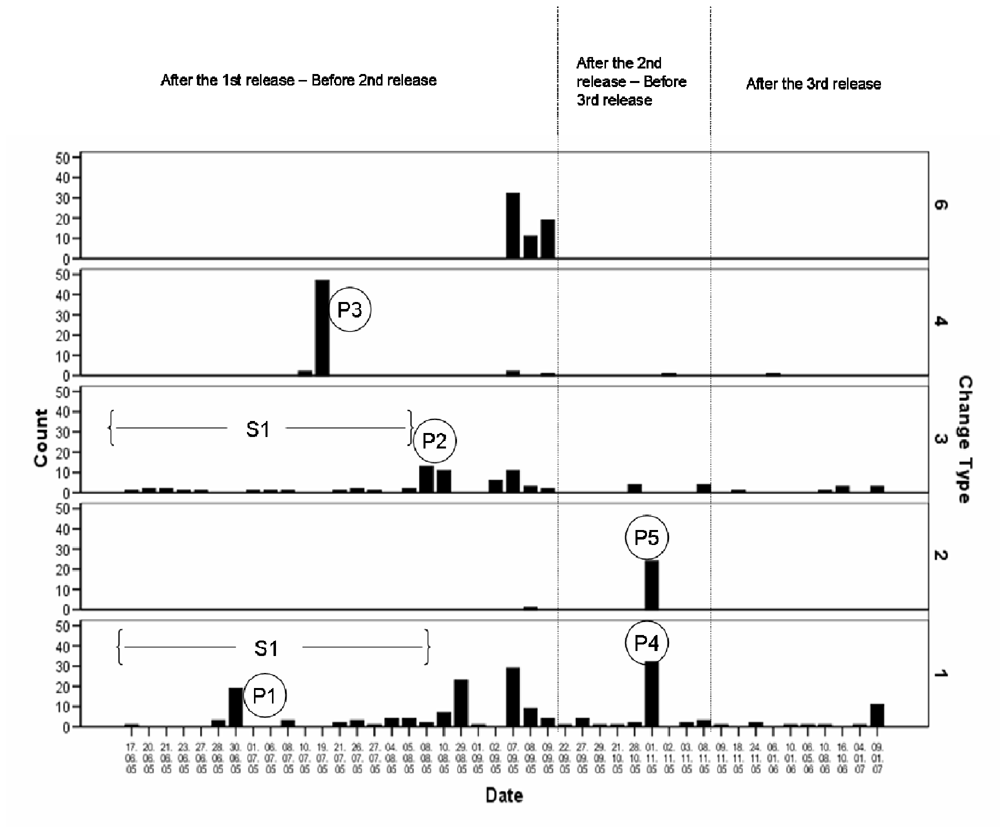

Fig. 3. JEF change profile
(1: perfective, 2: preventive, 3: corrective, 4: adaptive, 5: other, 6: inspection)

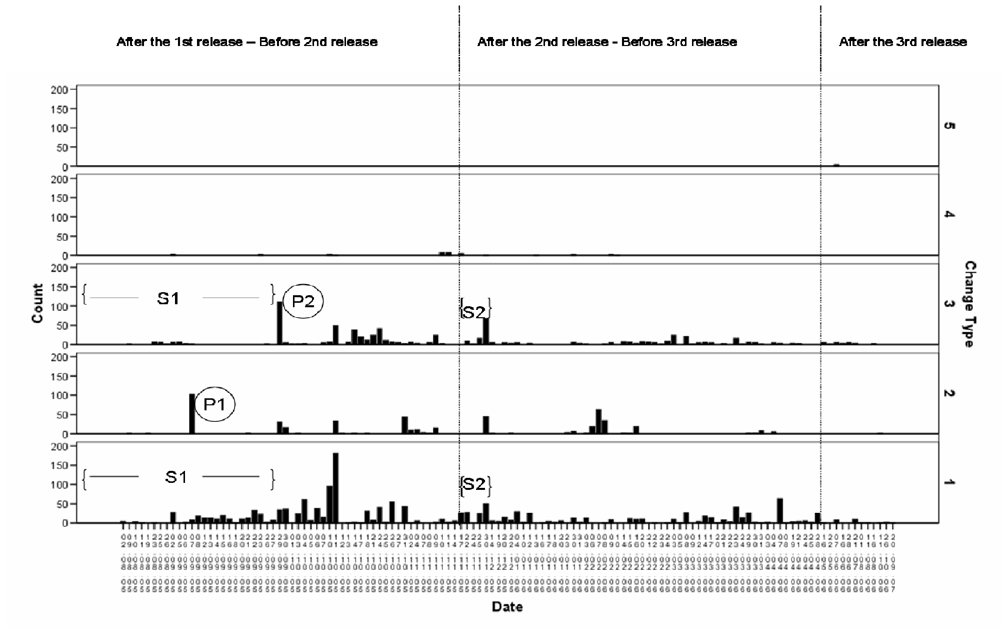

Fig. 4. DCF change profile
(1: perfective, 2: preventive, 3: corrective, 4: adaptive, 5: other)

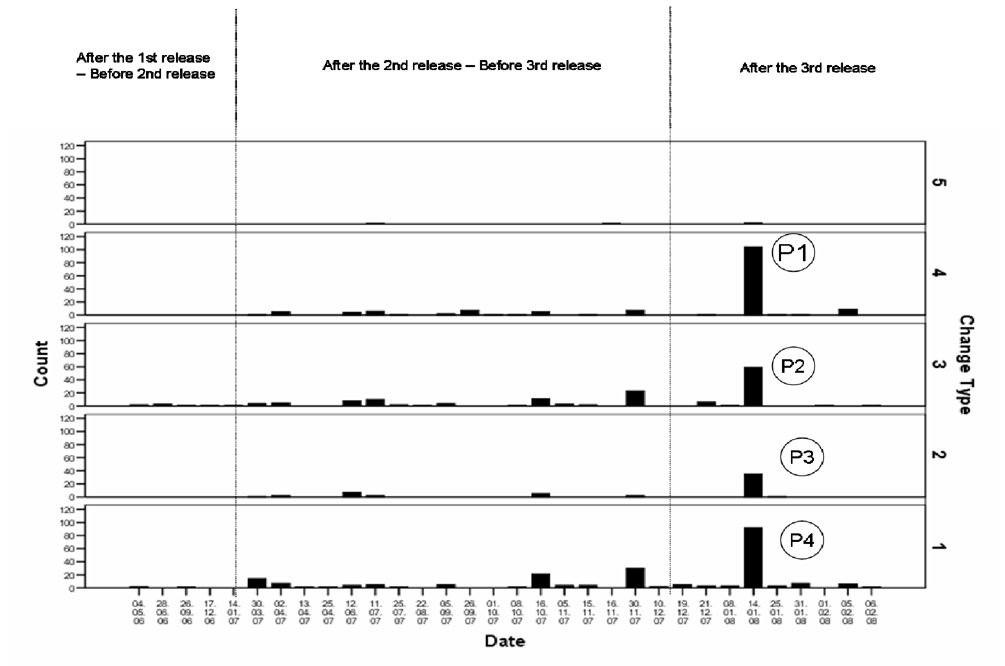

Fig. 5. S&A change profile
(1: perfective, 2: preventive, 3: corrective, 4: adaptive, 5: other)

The results from Fig. 3 to Fig. 5 reveal that:

- For the **JEF**, a few perfective and corrective changes were made after the first release was deployed (see stage S1 in Figure 3). However, perfective changes dominated during this stage, with a peak of perfective changes (see P1 in Figure 3) in June 2005. After stage S1, the number of corrective changes gradually increased, with a peak (see P2 in Figure 3) one month before the release date of the second release; then it decreased. Specifically, there were many adaptive changes (in July 2005) between the end of the first release and the deployment of the second release (see P3 in Figure 3). There was a peak of perfective changes (see P4 in Figure 3) and a peak of preventive changes (see P5 in Figure 3) in the middle between the second and third releases. However, few other changes were made during this period. After the third release, there were very few corrective and perfective changes.
- For **DCF**, the maintenance activity started with a few perfective changes after the first release (see stage S1 in Figure 4). Then, a corrective change peak occurred (see P2 in Figure 4). For preventative changes, there was a peak (see P1 in Figure 4) between the deployment of the first release and the second release in September 2005. After the deployment of the second release, the dominating changes were first perfective changes (see Stage S2 in Figure 4), followed by corrective changes. After the third release, very few changes were made. A few perfective and corrective changes were made.
- For **S&A**, few changes were made after the deployment of the first and second releases. After the deployment of the third release, several perfective, corrective, and preventative changes were made simultaneously (see P1 to P4 in Figure 5). Most changes were made after the deployment of the third release between December 2007 and January 2008.

Table 11. The root cause analysis for RQ2.1
C: Complexity, F: Functionality, D: Development practice, A: Age, S: Size

Observation	**Factor**					**Explanations**
	C	F	D	A	S	
Many perfective changes were made to the JEF in June 2005 (see P1 in Figure 3).				X		The JEF was first used by PDM, before being reused by DCF and S&A (see Figure 2). After PDM had used the JEF, several changes were made regarding the functionality of the JEF.
Many corrective changes were made to the JEF in August 2005 (see P2 in Figure 3).				X		The corrective changes were made as a result of the intensive testing of the JEF before clients started to use the application.
Many adaptive changes were made to the JEF in July 2005 (see P3 in Figure3).			X			The adaptive changes were due to changes in the version control system. The company changed their version control system from PVCS to Rational ClearCase in the middle of the project. All the files in the PVCS had a Java comment, but when the company changed to Rational ClearCase, the Java comments in all the files were removed.
Many perfective and preventive changes were made to the JEF in November 2005 (see P4 and P5 in Figure 3).			X			The perfective and preventive changes were due to the introduction of various third-party components.
Many preventive and corrective changes were made to DCF in Sept. 2005 (see P1 and P2 in Figure 4).			X			DCF was refactored in July 2005. The developers rewrote the whole DCFClient, which explains the large amount of preventive changes. Due to the refactoring, a new DCFClient was made. DCFClient then went through testing. That explains the high amount of corrective changes.
Most changes to S&A were made in December 2007 and January 2008 (see P1 to P4 in Figure 5).			X			After the deployment of Release 3, two new users saw the system and did a lot of acceptance tests. The new and changed requests from these new users and the results of new acceptance tests led to many changes of all types.

6 Discussion

6.1 The overall distribution of different types of change: both reusable and non-reusable software

Our results reveal that *perfective changes* are the most common for both the reused framework and applications reusing the framework. Although slightly different definitions of change types have been used here, our results seem to support the observations of [Lientz78][Jørgensen95][Evanco99][Satpathy02][Mohagheghi04][Lee05]. Several other studies [Burch97][Mockus00][Schach03] yielded different conclusions and showed that the majority of changes are either corrective or adaptive. However, the dominant type of change varies over time.

Our RCA shows that five factors, (i.e. complexity, functionality, development practice, age, and size) may determine the profile of software maintenance. From Tables 8-11 (counting the number of times each factor was chosen by the developer), we can see that the factors that affect maintenance the most in our case are *functionality* and *development practices*, followed closely by *complexity*. The factors that affect maintenance the least are *age* and *size*. In our case, the JEF received new requirements after it had been used by several applications, which led to many perfective changes being made during the maintenance and evolution phase. DCF faced high time-to-market pressure and had unclear requirements at the early stage of the development. For S&A, most perfective changes were made after two new users saw the system and introduced new requirements. Without these evolving requirements, perfective changes might not have dominated in the systems that we investigated.

The study of Schach et al. [Schach03] investigated three systems and observed that the dominant changes in their investigated systems are corrective, rather than perfective as in [Lientz78]. They explained the large amount of corrective changes by appeal to different programming domains, programming languages, and perspectives on study design. In our study, the dominant type of change was perfective, rather than corrective as was observed in [Scahc03]. This is despite the fact that our study had a design similar to [Schach03]. We also investigated systems that were programmed in object-oriented languages. The only difference in our investigated systems from those in [Schach03] was the programming domains. The products studied in [Schach03] include one commercial real-time product, the Linux kernel, and GCC. Our investigated systems and those systems studied in [Lientz78] are basically business and data processing software, which often face new or changed requirements from clients due to their changing business needs. This difference explains the fact that perfective changes dominated in the software studied in [Lientz78] and our study, and other types of changes dominated in the products examined in [Schach03]. Schach et al. [Schach03] indicated that few textbooks (only one of the three top-selling ones [Pressman01][Sommerville04][Schach02]) accept that the results of [Lientz78] cannot be extrapolated to all types of software.

6.2. The distribution of different types of change: comparison between reusable vs. non-reusable software

With respect to differences in change density between the reused framework and applications reusing it, our results show that the reused framework has a higher change density than one application and a lower change density than the other. These results which do not support conclusions from any previous studies [Frakes01][Algestam02][Mohagheghi04a][Selby05]. The RCA revealed that developers in the company were cautious about making changes to the JEF because the changes may affect existing applications. The change density of the reused framework may have been reduced as a result of this concern. On the other hand, it is impossible to predict all future requirements of a reusable framework. Unforeseen requirements of new applications may demand that many minor or major (such as refactoring) changes be made to the reusable framework. This could explain the higher change density of the JEF compared to S&A. Another observation of our study is that both the reused framework and applications follow the so-called “80/20” rules [Schaefer85][Kemerer97], i.e. about 80% of all work is caused by only 20% of all components. In our case, one or two components in each system covered most of the changes, such as JEFClient and JEFWorkBench in JEF; DCFClient and DCFCommon in DCF; SACommon and SAClient in S&A. The components that require the most changes are usually the most complex, largest, or the ones involving several GUIs.

6.3 Software change profile over time

It is important to estimate the change profile of a software system in order to arrange staff expertise, tools, and business strategies properly [Bennett00]. Compared with previous studies on software change profiles [Gefen96][Burch97][Bennett00], the change profiles of the JEF and DCF lend support to the simple/versioned stage model proposed by [Bennett00]. We did not measure user-support activities; hence, our study is not comparable with [Burch97].The pattern proposed in [Gefen96] does not fit our data, because the first stage of the software changes was not centered on corrective modifications. For DCF and the JEF, more perfective changes than corrective changes were made at the start of evolution and maintenance. Thus, the change profiles of the JEF and DCF are in line with the software lifecycle that is proposed in [Bennett00]: an initial development stage, followed by a stage to extend the software’s capabilities and functionalities to meet user needs, then a stage in which only minor defect repairs are made, and finally a phase-out stage. In addition, we noticed that the JEF arrived at the stage in which minor defect repairs are made faster than DCF. Very few changes occurred after the second release of the JEF, except for a peak of perfective changes and a peak of adaptive changes. By contrast, DCF still followed a simple stage model, where many perfective changes were made, followed by corrective changes after the second release. One explanation is that the JEF is a reusable framework. After it has been used and reused by several applications, the company is cautious about making more changes to it. The reason for this is that the changes that are made to the JEF affect several applications reusing it. Making changes to the JEF is therefore difficult and expensive. In another word, the reused software goes to “code decay” [Eick01] or “software rot” [Wiki08] more quickly than applications that reuse it.

The change profile of S&A is different from those of the JEF and DCF. All types of change occurred intensively right after the involvement of new clients. A possible explanation for this phenomenon is that the software is still in the stage in which its capabilities and functionality are being extended to meet user needs. We expect that after this stage is complete, there will be a repair stage, which will include making many corrective changes to correct defects created in the previous stage. Although the RCA shows that several of the phenomena of the change profiles of our investigated systems can be attributed to the development practice, we still presume that the dynamic profile of software change can be influenced by Kemerer’s five factors [Kemerer97].

6.4 Insights on the improvement of software reuse

Our results show both benefits and challenges of software reuse with respect to software evolution and maintenance. Reusable software may be more stable and need less maintenance effort, if the context of reuse can be predicted accurately. In addition, it is important to have a well-designed architecture to reduce the complexity of reusable software. For the long-term evolution and maintenance of reusable software, our results indicate that staff who understand reusable software well must be retained in the organization for a while after initial development. Such action is necessary because the software may experience a stage in which its capabilities and functionality are extended to meet user needs, which will require making many major changes, after the initial deployment. As proposed in [Bennett00], staff expertise is critical both during the initial development of reusable software and when its capabilities and functionalities are being extended to meet user needs. Once the reusable software is stable and is at a stage in which only minor defect repairs are required, little staff expertise is needed because no dramatic change is welcome after the software has been reused by many applications. The change profile of S&A indicates that, for all software (whether reusable or non-reusable), developers need to prepare for all kind of changes when new clients evaluate the software.

6.5 Possible threats to validity

We here discuss possible threats to validity in our case study, using the definitions provided by [Wohlin00]:

Construct Validity: The definitions of different types of change used in our study are slightly different from those used in some previous studies, as was discussed in Section 2. Thus, direct comparisons of our results with previous studies need to be discussed case by case. RCA is often performed on each change. One possible threat to construct validity is that we performed our RCA on a subset of all changes. Given that we did not perform a detailed analysis of each change, we may have missed important details. However, in StatoilHydro ASA several of the developers who are involved in the project are external consultants and when they have completed their work on the project, they leave. This made it difficult for us to trace all changes back to each developer. Therefore, we did not have the resources to perform a RCA of each change. Another possible threat to construct validity is that we asked only one senior developer about the cause of the changes during the RCA. RCA with other developers can further verify or falsify explanations made by this developer. However, as mentioned above, once the external consultants have finished their work on a project, they leave, so we could not find other people who knew the details of all the three systems that we investigated.

External Validity: The entire data set was taken from one company. The object of study was a class framework and only two applications. Further similar studies need to be performed in different contexts and organizations in order for our results to be generalizable.

Internal Validity: All of the software changes that we investigated were classified manually by us. The first and the second author of the paper classified all the changes separately and then cross-validated the results. This was to enhance the validity of the data. A threat to the internal validity is the number of files that we selected randomly from Rational ClearCase. The random sampling might have caused a systematic bias. Our random selection might have yielded files with few changes, while the files with the most changes were left behind.

Conclusion Validity: This study is an explorative case study without any testing of hypotheses. Thus, the threat to conclusion validity is low.

7. Conclusion and future work

Few published empirical studies have characterized and compared the software changes made to a reusable framework with those made to a non-reusable application from a longitudinal perspective. We have presented the results from a case study of software changes that were performed on a reusable class framework and two applications reusing it. We studied the change density, the distributions of different types of change, and their properties over time. Our results contribute to a deeper understanding of software change over time and the relation between software change and software reuse. We conclude that:

- Our results only partially support previous findings that reusable software is more stable than non-reusable software. Factors that help to reduce of the number of changes that need to be made to reusable software are good initial design and stable dependence on the part of software reusing it. The prime factor that may increase the number of changes that are made to reusable software is unpredictable contexts of usage.
- The change profile of the systems that we investigated during the maintenance and evolution phases usually goes as follows: initial development, followed by a stage in which the system’s capabilities and functionality are extended to meet user needs, then a stage in which only minor defect repairs are made, and finally a phase-out stage. Reused software goes from extending the capabilities and functionality to minor defect repairs much faster then non-reusable software.
- The factors that affect maintenance the most in our case are software functionality, development practices, and software complexity. Other factors, such as software age and software size, also need to be considered when predicting software maintenance effort and when performing and presenting studies on software maintenance. However, further studies over a spectrum of application domains are required if their precise role and impact are to be determined.

The results of our study are relevant to industrial practitioners in that they show that more systematic reuse policies need to be developed and followed if the change density of reusable software is to be reduced. For researchers, the results indicate a set of diverse factors to be studied and considered when discussing the relation between software reuse and software evolution and maintenance.

This study reports on only three systems in one company over three years and the results are exploratory. It is our intention to collect data on software change from more companies to validate our conclusions and to build a model to predict software maintenance and evolution activities and effort.

Acknowledgement

This work was financed in part by the Norwegian Research Council for the SEVO (Software EVOlution in Component-Based Software Engineering) project [SEVO04] with contract number 159916/V30. We would like to thank StatoilHydro ASA for the opportunity for involving us in their reuse projects and senior developer Thor André Aresvik for valuable comments.

References

[Abran91] A. Abran, H. Nguyenkim, Analysis of Maintenance Work Categories Through Measurement, Proceedings of the IEEE Conference on Software Maintenance, IEEE Computer Society Press, Sorrento, Italy, 1991, pp. 104-113.

[Algestam02] H. Algestam, M. Offesson, L. Lundberg, Using components to increase maintainability in a large telecommunication system, Proceedings of the Ninth IEEE Asia-Pacific

Software Engineering Conference, IEEE Computer Society Press, Gold Coast, Australia, 2002, pp. 65-73.

[Basili96] V. Basili et al., Understanding and Predicting the Process of Software Maintenance Releases, Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society Press, Berlin, Germany, 1996, pp. 464-474.

[Bennett00] V.T. Rajlich, K.H. Bennett, A Staged Model for the Software Life Cycle, IEEE Computer, 33 (2000) 66-71.

[Burch97] E. Burch, H.J. Kung, Modeling Software Maintenance Requests: A Case Study, Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Bari, Italy, 1997, pp. 40-47.

[Card98] D.N. Card. Learning from Our Mistakes with Defect Causal Analysis. IEEE Software. 15 (1998) 56-63.

[Chapin85] N. Chapin, Software maintenance: a different view, In AFIPS Conference Proceedings, Volume 54 NCC, AFIPS Press, Reston, Virginia, 1985, pp. 328-331.

[Chapin01] N. Chapin, et al., Types of Software Evolution and Software Maintenance, Journal of Software Maintenance and Evolution: Research and Practice, 13 (2001) 3-30.

[Eick01] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, A.Mockus, Does Code Decay? Assessing the Evidence from Change Management Data, IEEE Transactions on Software Engineering, 27(2001) 1-12.

[Evanco99] W.M. Evanco, Analyzing change effort in software during development, Proceedings of the 6th IEEE International Software Metrics Symposium, IEEE Computer Society Press, Boca Raton, Florida, 1999, pp. 179-188.

[Fenton96] N. E. Fenton and S. L. Pfleeger, Software Metrics, PWS Publishing Company, 1996.

[Frakes01] W.B, Frakes, An industrial study of reuse, quality, and productivity, Journal of Systems and Software, 57 (2001) 99-106.

[Gefen96] D. Gefen and S. L. Schneberger, The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications, Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Monterey, California, 1996, pp 131-141.

[Jørgensen95] M. Jørgensen, The quality of questionnaire based software maintenance studies, ACM SIGSOFT – Software Engineering Notes, 20 (1995) 71-73.

[Kemerer97] C. F. Kemerer, S.A. Slaughter, Determinants of Software Maintenance Profiles: An Empirical Investigation, Journal of Software Maintenance, 9 (1997) 235-251.

[Kitchenham99] B. A. Kitchenham et al., Towards an Ontology of Software Maintenance, Journal of Software Maintenance: Research and Practice, 11 (1999) 365-389.

[Krogstie06] J. Krogstie, A. Jahr, D.K. Sjøberg, A longitudinal study of development and maintenance in Norway: Report from the 2003 investigation. Information and Software Technology, 48 (2006) 993-1005.

[Lee05] M.G. Lee, T.L. Jefferson, An Empirical Study of Software Maintenance of a Web-based Java Application, Proceedings of the 21st IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Budapest, Hungary, 2005, pp. 571-576.

[Lehman80] M.M. Lehman, Programs, Life Cycles and Laws of Software Evolution, Proc. Special Issue Software Eng., 68 (1980) 1060-1076.

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

[Lim94] W. Lim, Effect of Reuse on Quality, Productivity and Economics, IEEE Software, 11 (1994) 23-30.

[Macorr08] Marketing Correlation, 2008, http://www.macorr.com/ss_calculator.htm

[Mockus00] A. Mockus, L.G. Votta, Identifying Reasons for Software Changes Using Historical Databases, Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society Press, San Jose, California, 2000, pp. 120-130.

[Mohagheghi04a] P. Mohagheghi, R. Conradi, O.M. Killi, H. Schwarz, An Empirical Study of Software Reuse vs. Defect Density and Stability, Proceedings of the 26th IEEE International Conference on Software Engineering, IEEE Computer Society, Edinburgh, Scotland, 2004, pp. 282-292.

[Mohagheghi04b] P. Mohagheghi, R. Conradi, An Empirical Study of Software Change: Origin, Impact, and Functional vs. Non-Functional Requirements, Proceedings of the IEEE International Symposium on Empirical Software Engineering, IEEE Computer Society Press, Redondo Beach, Los Angeles, 2004, pp. 7-16.

[Pigoski97] T. M. Pigoski, Practical Software Maintenance, Wiley Computer Publishing, 1997.

[Postema01] M. Postema, J. Miller and M. Dick, Including Practical Software Evolution in Software Engineering Education, Proceeding of the 14th Conference on Software Engineering Education and Training, IEEE Computer Society, Press, Charlotte, North Carolina, 2001, pp. 127-135.

[Pressman01] R.S. Pressman, Software Engineering, A Practitioner's Approach, McGraw-Hill, Boston, 2001.

[Satpathy02] M. Satpathy, N.T. Siebel, D. Rodríguez, Maintenance of Object Oriented Systems through Re-engineering: A Case Study, Proceedings of the 10th IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Montreal, Canada, 2002, pp. 540-549.

[Schach02] S. R. Schach, Object-Oriented and Classical Software Engineering, McGraw-Hill, Boston, 2002.

[Schach03] S.R. Schach, B. Jin, L. Yu, G. Z. Heller, J. Offutt, Determining the Distribution of Maintenance Categories: Survey versus Management, Journal of Empirical Software Engineering 8 (2003) 351-366.

[Schaefer85] H. Schaefer, Metrics for optimal maintenance management, Proceedings Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA, 1985, pp. 114-119.

[Selby05] W. Selby, Enabling Reuse-Based Software Development of Large-Scale Systems, IEEE Transactions on Software Engineering, 31(2005) 495-510.

[SEVO04] The Software EVOlution (SEVO) Project, 2004-2008, http://www.idi.ntnu.no/grupper/su/sevo/

[Sommerville04] I. Sommerville, Software Engineering, Addison-Wesley, UK, 2004.

[Sousa98] M. Sousa, H. Moreira, A Survey on the Software Maintenance Process, Proceedings of the 14th IEEE International Conference on Software Maintenance, IEEE Computer Society Press, Bethesda, Maryland, 1998, pp. 268-274.

[Swanson76] E.B. Swanson, Proceedings of the Second IEEE International Conference on Software Engineering, IEEE Computer Society Press, San Francisco, California, 1976, pp. 492-497.

[Wiki08] Wikipedia on Software rot, 2008, http://en.wikipedia.org/wiki/Software_rot

[Wohlin00] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering – An Introduction, Kluwer Academic Publishers, 2002.

[Yip94] S. Yip, T. Lam, A Software Maintenance Survey, Proceedings of the First IEEE Asia-Pacific Software Engineering Conference, IEEE Computer Society Press, Tokyo, Japan, 1994, pp. 70-79.

[Zhang05] W. Zhang, S. Jarzabek, Reuse without compromising performance: industrial experience from RPG software product line for mobile devices, Proceedings of the 9th International Software Product Line Conference, Springer, Rennes, France, 2005, pp. 57-69.

Appendix A: Detailed information of the investigated systems

The JEF is designed on the basis of a technical architecture for all J2EE systems in the company. This architecture has four logical layers. The following presentation describes the layers from top to bottom:

(1) Presentation: Responsible for displaying information to the end user and to interpret end-user input.

(2) Process: Provides support for the intended tasks of the software and configures the domain objects.

(3) Domain: Responsible for representing the concepts of the business and information about the business and business rules. This layer is the heart of the system.

(4) Infrastructure: Provides general technical services such as transactions, messaging, and persistence.

The components in this framework are built from a combination of COTS (Commercial-Off-the-Shelf) components, OSS (Open Source System) components, and some code that was developed in-house. Table A1 gives information about the JEF, using Kitchenham's ontology of software maintenance as a basis[Kitchenham99].

Table A1. Background information for the JEF

Ontology	**Description**
Product size	– Release 1: 14.June 2005: 16875 NSLOC. – Release 2: 9.Sept. 2005: 18599 NSLOC. – Release 3: 8.Nov.2005: 20348 NSLOC. – One development team.
Application Domain	Java technical framework for developing Enterprise Applications.
Product Age	3.5 years
Product Maturity	Adolescence
Product Composition	Consists of seven separate components, which can be applied separately or together when developing applications.
Product and Artefact Quality	Well-defined requirement and design document.
Development Technology	– J2EE (Java 2 Enterprise Edition). – SPRING framework
Paradigm	– Object-Oriented paradigm (Java). – Design patterns.

	– Partially Open-Source Development.
Maintenance Organization Process	– Software development plan. – Software configuration management tool.

DCF is used mainly for storing documents. It imposes a certain structure on the documents stored in the application and relies on the assumption that the core part of the documents is based on cargo (load) and deal (contract agreement) data, as well as auxiliary documents that pertain to this information. DCF is meant to replace the current handling of cargo files, which are physical folders that contain printouts of documents that pertain to a particular cargo or deal. A cargo file is a container for working documents that are related to a deal or cargo, within operational processes, used by all parties in the oil sales, trading, and supply strategy plan of the company. There are also three releases of the DCF application. Table A2 gives information about DCF.

Table A2. Background information for DCF

Ontology	**Description**
Product size	– Release 1: 1. Aug. 2005: 20702 NSLOC. – Release 2: 14.Nov. 2005: 21459 NSLOC. – Release 3: 8.May 2006: 25079 NSLOC. – One development team.
Application Domain	A document storage application to manage cargo files. A cargo file is a container for working documents that are related to a deal or cargo, within operational processes, used by all parties in the company
Product Age	3.5 years.
Product Maturity	Senility (legacy).
Product Composition	Eight components built in-house, Biztalk, MessageManager, and Meridio.
Product and Artifact Quality	Poor requirement and design document in the beginning.
Development Technology	– J2EE (Java 2 Enterprise Edition). – SPRING framework.
Paradigm	– Object-oriented paradigm (Java). – Design patterns.
Maintenance Organization Process	– Software development plan. – Software configuration management tool.

S&A is an application for controlling business processes and carrying them out more efficiently through common business principles within lift and cargo planning. Lift planning is based on a lifting program, which is the function area for generating an overview of the cargoes that are scheduled to be lifted. The lifting program operates on a long-term basis (e.g. 1 - 12 months), and generates tentative cargoes mainly on the basis of closing stock and predictions about production. The lifting program is distributed to the partners so that they can plan how they will handle the lifting of their stock. The cargo planning and shipment covers activities to accomplish the lifting. Input to the process is the lifting program. While carrying out the process, users will enter detailed information about the cargo on the basis of document instruction from partners and perform short-term planning on the basis of pier capacity and storage capacity. After loading, sailing telex and cargo documents are issued. Then the cargo is closed and verified. The S&A application allows the operators to carry out "what-if" analysis on shipments that are to be loaded at terminals and offshore. Given that the current system ("SPORT") is not able to handle complex agreements (i.e. the mixing of different qualities of oil within the same shipment), it allows the transfer and entry of related data, which is currently often done manually, to be digitized and automated. The main goal of the S&A application is to replace some of the current processes/systems, in addition to providing new functionality. The S&A application has also three releases. Table A3 gives information about S&A.

Table A3. Background information for S&A

Ontology	**Description**
Product size	– Release 1: 2 May 2006: 29957 NSLOC. – Release 2: 6.Feb. 2007: 50879 NSLOC. – Release 3: 12.Dec. 2007: 64319 NSLOC. – One development team.
Application Domain	An application for controlling and performing business processes more efficiently through common business principles within lift and cargo planning.
Product Age	2.5 years.
Product Maturity	Senility (legacy).
Product Composition	Seven components built in-house.
Product and Artifact Quality	Well-defined requirement and design document.
Development Technology	– J2EE (Java 2 Enterprise Edition). – SPRING framework.
Paradigm	– Object-oriented paradigm (Java). – Design patterns.
Maintenance Organization Process	– Software development plan. – Software configuration management tool.

Appendix B: Secondary papers

In this Appendix we have included two papers. SP1 is considered to be outside the scope of this thesis, and SP2 is overlapping with the paper P5 in Appendix A. The papers are:

- **SP1:** *An Empirical Study of Distributed Technologies Used in Collaborative Tasks at Statoil ASA*
- **SP2:** *The Empirical Studies on Quality Benefits of Reusing Software Components*

SP1: An Empirical Study of Distributed Technologies Used in Collaborative Tasks at Statoil ASA

Anita Gupta, Marianne H. Asperheim and
Odd Petter N. Slyngstad
Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway

Harald Rønneberg
Statoil KTJ/IT
Stavanger (Forus), Norway

Published: In Enrico Blanzieri and Tao Zhang (Eds.): Proc. 2nd International Conference on Collaborative Computing (CollaborateCom'06), 17-20 November 2006, Atlanta, Georgia. IEEE CS Press, ISBN 1-4244-0429-0, pp. 1-5.

Abstract. This paper presents results of a survey, related to the theoretical Task-Technology-Fitness framework. The survey was conducted in a large Oil and Gas company in Norway, namely Statoil ASA. The Task-Technology-Fitness framework indicates which groups of medium or technology are appropriate to choose according to the task to be performed, when collaborating with others. We have here presented the extended version of the Task-Technology-Fitness framework, according to how Statoil ASA's employees use SMS, e-mail, Instant Messaging and Audio (phone call), in different collaborative tasks. In total, there were 333 out of 747 respondents who participated in the survey. The results reveal that SMS and Instant Messaging are not seen as efficient or well suited communication channels for collaborative tasks. E-mail seems to be favorable among the respondents for the collaborative tasks, while audio (phone call) follows closely. The results are important in that they indicate when SMS, e-mail, Instant Messaging and Audio (phone call) are appropriate to use. The purpose of the survey was to discover potential area of improvements for Statoil ASA.

SP2: Empirical Studies on the Quality Benefits of Reusing Software Component

(Position paper)

Jingyue Li, Anita Gupta, Jon Arvid, Børretzen, and Reidar Conradi
Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
{jingyue, anitaash, borretze, conradi}@idi.ntnu.no

Published: In Xiaodong Liu et al. (Eds.): Proc. 1st International Workshop on Quality Oriented Reuse of Software (QUORS'07), 23-27 July 2007, Beijing, China. IEEE CS Press, ISBN 978-0-7695-2870-0, pp. 399-402.

Abstract. The benefits of reusing software components have been studied for many years. Several previous studies have concluded that reusing components have fewer defects in general than non-reusable components. However, few of these studies have gone a further step, i.e., investigating which type of defects has been reduced because of reuse. Thus, it is suspect that making a software component reusable will automatically improve its quality. This paper presents an on-going industrial empirical study on the quality benefits of reuse. We are going to compare the defects types, which are classified by ODC (Orthogonal Defect Classification), of the reusable component vs. the non-reusable components in several large and medium software systems. The intention is to figure out which defects have been reduced because of reuse and the reasons of the reduction.