Component-Based Software Engineering: Modern Trends, Evolution and Perceived Architectural Risks

Odd Petter Nord Slyngstad

Doctoral Thesis

Submitted for the Partial Fulfilment of the Requirements for the Degree of

Philosophiae Doctor

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

April 2011

(This page will also be removed by NTNU-trykk and replaced with a template. This page is known as the colophon page, or kolofon side)

You will need the two ISBN numbers and the internal NTNU "thesis number" that year. All this can be obtained from http://ojapp01.itea.ntnu.no:7780/isbnprovider/start.do

The ISSN serial number is the same for all doctoral theses at NTNU, and is 1503-8181.

ISBN (printed version) 978-82-471-2704-9
ISBN (electronic version) 978-82-471-2705-6
ISSN 1503-8181
Thesis at NTNU 2011:87

Printed in Norway by NTNU Trykk, Trondheim

…og bakom duver blånene…

Abstract

Motivation: Component-Based Software Engineering (CBSE) is an approach to software reuse where software assets or artifacts from multiple sources are reused to develop systems faster and cheaper. The main benefit of software reuse in CBSE is the enabling of systematic improvement in terms of quality, effort, time-to-market, common software platform/architecture, and standards compliance. A key aim of these improvements is to enable proper management of CBSE-driven software evolution, i.e. helping software engineers become more cost-effective in developing and incorporating high-quality, reusable components and other assets.

Knowing the relevant risks and effective handling strategies in software evolution is paramount to achieving improvements in quality, effort and time-to-market. Moreover, the architecture of a software system constitutes its fundamental building blocks. Continued suitability of the architecture over time is therefore crucial to the continued success of the system. Prior investigations on risks and risk management strategies have commonly focused on project-level risks and strategies. Similarly, studies on software architecture have mainly investigated its design, implementation and maintenance. Little prior effort has been made towards studying risks and risk management strategies of architectural evolution.

Approach: This thesis investigates the state of practices and issues of modern CBSE, with multi-origin reusable components (in-house software, Commercial Off-The-Shelf software (COTS), and Open Source Software (OSS)), in the development process, based on quantitative and qualitative empirical studies of industrial systems. It also explores software evolution impact, elicited through defect and change reports over time. Test Driven Development (TDD) as a strategy to handle these impacts is also investigated. Finally, surveys are performed on risks and risk management strategies in industrial software projects.

The **aims** (research questions) in this thesis are:

RQ1: *What is the state of practices and issues with respect to software process improvement in CBSE for COTS/OSS and in-house reusable software?* This is answered by two industrial surveys.

RQ2: *How does software evolution impact individual reusable components, in terms of defect and change densities?* This is answered by an industrial case study.

RQ3: *What are the impacts of Test Driven Development versus test-last development on reusable components?* This is answered by an industrial case study.

RQ4: *What are the perceived architectural risks of CBSE-driven software evolution, and how can these risks be mitigated?* This is answered by two industrial surveys.

Correspondingly, the **contributions** of this thesis are (elaborated in articles P1-P6):

C1. *Improved knowledge of modern trends in CBSE and their impacts on software development processes (RQ1, articles P1, P2).*

C2. *Improved understanding of evolution impact on individual reusable components in terms of defect and change densities (RQ2, article P3).*

C3. *Improved understanding of the impact and effectiveness of TDD (RQ3, article P4).*

C4a. *Identification of perceived risks and related mitigation strategies specifically for the evolution of software architecture (RQ4, articles P5, P6).*

C4b. *An adapted operational matrix as a tool to support risk management in software architecture evolution (RQ4, article P6).*

Preface

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

The work referred to has been performed at the Department of Computer and Information Science, NTNU, Trondheim, under the main supervision of Dr. Reidar Conradi.

The thesis was financed in part through a fellowship stipend provided by the Research Council of Norway (3 years), as part of the SEVO (Software EVOlution in Component-Based Software Engineering) project with contract number 159916/V30. In addition to the PhD fellowship, I have been working one year as a teaching assistant at NTNU.

Acknowledgements

This thesis comprises a part of the SEVO project (Software EVOlution in Component-Based Software Engineering). The data collection was performed partly at StatoilHydro ASA at Trondheim/Stavanger, and partly in the European IT-industry.

Cooperation with other persons is a key part of being able to complete work on a PhD. The environment I've been allowed to be a part of has fostered independent thought along with constructive cooperation. I'd like to thank my main surpervisor, Dr. Reidar Conradi, for continuous constructive advice and support during my doctoral work. I'd also like to thank Dr. Jingyue Li, Dr. Letizia Jaccheri, Dr. Tor Stålhane, and all the other members of the Software Engineering group at IDI/NTNU. I'd also like to recognize the importance of international collaboration on publications I was allowed to be a part of through my colleagues from Germany (Dr. Bunse), Italy (Dr. Morisio, Dr. Torchiano), China (Dr. Jingyue Li), Ireland (Dr. Babar) and the Netherlands (Dr. van Vliet).

Thank you also to Harald Rønneberg, Einar Landre, Harald Wesenberg and everyone else I've been involved with at StatoilHydro ASA. I'd also like to thank all my colleagues with whom I've had the pleasure to share and exchange knowledge and experiences.

Finally, I want to thank Sari for her love and continuing support throughout my PhD. I'd also like to thank my friends and family for their support and encouragement.

IDI, NTNU, April, 2011

Odd Petter Nord Slyngstad

Contents

List of Figures

List of Tables

Abbreviations

ACM	Association for Computing Machinery
ALMA	Architecture-Level Maintainability Analysis
ASA	Norwegian for "Incorporated"
ATAM	Architecture Tradeoff Analysis Method
CBSE	Component-Based Software Engineering
COM	Common Object Model
COTS	Commercial Off-The-Shelf software/components in this thesis (also appears as "Components Off-The-Shelf" in articles P1-P6, which denotes the same concept).
CR	Change Request
DCF	Digital Cargo Files software system at StatoilHydro ASA
FAQ	Frequently Asked Questions
FEAST	Title of two research projects under the direction of Dr. M. Lehman
IBM	International Business Machines, Inc.
IDI	Department of Computer and Information Science at the Norwegian University of Science and Technology
IEEE	Institute of Electrical and Electronics Engineers, Inc.
ISO	International Standards Organization
IT	Information Technology
J2EE	Java 2 Enterprise Edition
JEF	Reusable framework of components based on Java Enterprise Framework components at StatoilHydro ASA
MFC	Microsoft Framework Component
NATO	North Atlantic Treaty Organization
NSLOC	Non-commented Source Lines of Code
NTNU	Norwegian University of Science and Technology
OO	Object Oriented
OR	Organizational Risk
OS	Organizational Strategy
OSS	Open Source Software
OTS	Off-The-Shelf software components, including COTS and OSS components
PR	Process Risk
PS	Process Strategy
SAAM	Software Architecture Analysis Method
SEVO	Software EVOlution project, supported by the Research Council of Norway
SPI	Software Process Improvement
SEI	The Software Engineering Institute (SEI) at Carnegie Mellon University
SWOT	Strengths, Weaknesses, Opportunities and Threats analysis
TDD	Test Driven Development
TR	Articles P1, P3, P4: Trouble Report. Articles P5, P6: Technical Risk
TS	Technical Strategy

1 Introduction

This chapter summarizes the research background and context for this thesis. Additionally, it includes a description of the research questions, research design and corresponding contributions. Finally, it presents a list of the publications included and an outline of the remainder of the thesis.

1.1 Problem Outline

The SEVO (Software EVOlution) research project, funded by the Research Council of Norway from 2004 – 2008, defined the following goals that function as overall aims for this thesis:

- G1. Better understanding of software evolution, especially for Component-Based Software Engineering (CBSE).
    - To understand the state-of-practice and issues of CBSE in individual companies as well as in the IT-industry at large:
        - Modern trends in CBSE technology enable main advantages of software reuse. These advantages include improvements in quality, effort (cost) and time-to-market, and standards [Sommerville 2010].
        - Software evolution is inevitable in any software system since changes in society and technology will require subsequent changes to software systems to keep them up to date [vanVliet 2008]. Moreover, efficiency in the software process is paramount due to the ever-increasing demand on available development capacity. The cooperation necessary in software engineering impacts e.g. the distribution of work, communication, standards and procedures.
        - A related theme is the increased usage of Commercial Off-The-Shelf components and Open Source Software (OSS) in new development. These have different characteristics than in-house developed non-reusable components due to e.g. vendor control, selection and integration issues [Li 2004]. Their impact on the development and maintenance processes is therefore also different.
- G2. Better methods to predict the risks, costs, and profile of software evolution in CBSE.
    - To understand CBSE in detail on the software component/software system and software process levels in order to develop solutions to these issues:

        - Software Process Improvement (SPI) [Aaen 2001], i.e. systematically incorporating these solutions as part of revised development practices, is key. This will enable software engineers, software designers, software architects and the like to improve their cost-effectiveness in developing quality software based on reusable components. It will also enable them to improve their ability to develop and use reusable assets, such as code and process models.
        - Risks and risk management strategies are closely related to the above-mentioned (G1) issues in CBSE [Boehm 1991]. Knowledge of the impact and effectiveness of both risks and risk management strategies in CBSE-driven software evolution is paramount to the success of the key points mentioned above.
        - Software architecture constitutes a central part of any software system [Bass 2004], and is also an important concern seen in our investigation [P1]. We must therefore pay close attention to the design, maintenance and evolution of the architecture, to secure the continued success of the system. Awareness of potential architectural evolution risks is important as architectural changes can permeate a software system.
        - On the one hand, earlier investigations in risk management have commonly focused on risks and risk management strategies on the project level. On the other hand, software architecture investigations commonly study the design, implementation and maintenance of the architecture [P5]. Little prior effort has been made in the direction of studying risks and risk management strategies in direct relation to software architecture and its evolution. Moreover, earlier studies in this area have focused on output from structured evaluations of the architecture, while the actual methods used to evaluate architecture in industry can range quite widely [Babar 2007a]. Context-specific factors such as the physical size of the personnel groups used in architecture evaluation may also have an influence [Babar 2007b]. Further improvements are also needed with respect to the integration of architectural activities, notations and artifacts into software processes and tools [Buchgeber 2008].
- G3. Contributing to a national competence base in empirical software engineering.
    - There is the need for validated evidence to support or reject existing and revised hypotheses, models, design decisions, and the like within software engineering. Experience from empirical studies in the field can be incorporated into a living, experience-based knowledge base for use by the software engineering community.
- G4. Dissemination and exchange of the knowledge gained.
    - Active participation in, and publication to, peer reviewed venues (e.g. workshops, conferences and journals) are key not only to disseminate and exchange knowledge, but also to obtain knowledge on related work.

In this thesis, we first investigate the state-of-practice in CBSE, and thereafter use the knowledge gained towards investigating CBSE issues on a more detailed level.

Concretely, we investigate how modern trends in CBSE influence the software development and maintenance processes for COTS, Open Source and in-house reusable components (RQ1). Secondly, we use the results from the first investigation towards investigating the evolutionary trends of defect and change densities specifically for individual reusable components (RQ2). Finally, we investigate Test Driven Development (TDD) as a concrete example of software process improvement to handle risks in a specific company (RQ3), as well as perceived architectural risks pertaining to CBSE-driven software evolution in the IT-industry at large (RQ4).

1.2 Research Context

At the start of the project, we explored a number of different venues to obtain empirical data from the Norwegian IT-industry. Using contacts already established by our research group, we became involved with several software development projects that unfortunately did not yield usable research results. This was either due to lack of commitment or internal turmoil on the part of the industrial organizations in question, or lack of completeness in the data available. Nevertheless, after an upstart of half a year we were allowed to successfully follow two industrial projects for this thesis work.

Problems with respect to sample selection and response rates for the industrial surveys were also encountered. These issues are discussed in detail in the secondary article SP5 (Appendix B). In general, the software industry appears busy and without much time for participating in research efforts.

This thesis utilizes the results from quantitative as well as qualitative studies, using empirical data accumulated from software systems at StatoilHydro at two locations in Norway: Trondheim (Rotvoll) and Stavanger (Forus). In 2003, StatoilHydro started its own reuse program. Since 2004, this has become based on an in-house customized framework of reusable components, called the "JEF framework" (the name 'JEF' signifies that it is based on Java Enterprise Framework components). The aim of StatoilHydro was to explore the potential benefits of reusing software in a systematic manner. The framework is based on J2EE, and is a Java technical framework for developing Enterprise Applications [JEF 2006]. This initiative was started, as a response to changing business and market trends, by providing a shared platform for further in-house development and integration. The strategy is also being propagated throughout the company. Upper management has mandated that the JEF components are to be reused in all new development where applicable. There is also a training program in place for developers to gain knowledge of these components, their functionalities and their interfaces. The framework itself, and the corresponding applications that are using the framework, have been further developed in several releases over four years (2004 – 2008), with the latter releases being developed using Test Driven Development. Our research group was allowed to participate in the process to potentially verify some of the benefits of reuse sought by the company. The majority of the quantitative data comes from the ClearCase/ClearQuest software change management system, which is commonly in use at StatoilHydro. This system allows reporting of both Change Requests (for non-corrective changes) and Trouble Reports (for corrective changes or defects).

Empirical research is the result of thorough collaboration, and likewise so are the investigations leading up to this thesis. Thus, the results pertaining to overall defect and

change profiles for the various releases of the reusable and non-reusable software have been reported in the secondary articles SP8 and SP9, as well as in [Gupta 2009b]. The major difference is that the results from this investigation that are reported in this thesis pertain to the individual reusable components, rather than the overall releases of reusable software. Furthermore, the results from article P1 have previously been presented in [Gupta 2009b] with the aim to investigate possible improvements to the actual reuse practice at StatoilHydro ASA. This is also different from the work in this thesis, where we use the results from article P1 to investigate the impact of modern trends in CBSE on the development process for in-house reusable components, in comparison with the related impact for COTS/OSS components.

Additionally, this thesis uses both qualitative and quantitative data from other IT-companies. These data complement and further explore the results obtained from the above-mentioned data. In particular, data were elicited from a broad range of companies to obtain information on modern trends, evolution and software architectures in relation to CBSE. All these companies wish to remain anonymous, and are therefore not mentioned by name in this thesis.

1.3 Research Questions and Research Design

The thesis encompasses several empirical studies on the software-intensive industry, on both in-house and COTS/OSS-based development, all aimed at the main goals of the SEVO project as outlined above. The main focus is on CBSE as an overall umbrella; the need for upgrading development processes based on

1) modern trends in CBSE (RQ1),
2) metrics of change (RQ2),
3) impact of a specific SPI (i.e. TDD) (RQ3),
4) perceived architectural risks (RQ4).

Modern trends within CBSE have influenced development processes to evolve from the simple waterfall model towards more agile and flexible processes [Sommerville 2010], including development of COTS/OSS components. These new processes have become widely adopted, presenting new opportunities but also new risks [SP1, Appendix B], such as inaccurate effort estimation in project planning, and negative effects of component integration. There is also a lack of empirical validation of current methods for risk management issues in COTS/OSS development [SP1, Appendix B]. As an example, six commonly held facts about COTS development were debunked by Torchiano et al. [Torchiano 2004]. While results from P1 were also reported in [Gupta 2009b] to investigate possible improvements to the actual reuse practice at StatoilHydro ASA, it is therefore also important to study modernized processes and process changes for COTS/OSS and in-house reusable components. ***RQ1*** *is therefore defined as:*

What is the state of practices and issues with respect to software process improvement in CBSE for COTS/OSS and in-house reusable software?

When it comes to change metrics, we consider defect density (towards reliability) and change density (towards maintainability or evolution) to be important indicators of evolution in relation to the overall focus on CBSE. Earlier research on defect density showed that reusable components have lower defect densities across releases [Mohagheghi & Conradi 2004b]. Changes have previously been studied in terms of type and number rather than density [Mohagheghi & Conradi 2004a]. Further investigations

of these metrics with respect to overall releases of non-reusable versus reusable software have been explored by us in [SP8] [SP9], and are also reported in [Gupta 2009b]. However, gaining knowledge of software evolution impact on the level of individual reusable components is also important, and will allow us to propose further targeted handling of development issues in evolving industrial software components and CBSE systems. So ***RQ2*** *becomes:*

How does software evolution impact individual reusable components, in terms of defect and change densities?

Test Driven Development is an example of a software process improvement introduced to manage risks associated with CBSE-driven software evolution. Prior industrial studies on TDD, although few in number, have shown quality improvements (i.e. fewer corrective changes) but also a decrease in productivity compared to test-last development – for non-reusable components [Janzen 2005]. Prior studies appear to have neither investigated TDD's effect on non-corrective changes, nor its effect on reusable components. Reusable components may need to be more predictable, stable and maintainable [Mohagheghi & Conradi 2004a]. Thus, investigating TDD's impacts on reusable components is important to determine its effectiveness. So ***RQ3*** *becomes:*

What are the impacts of Test Driven Development versus test-last development on reusable components?

Finally, the continued operative success of the software architecture is paramount to the successful evolution of the corresponding software system. Bass et al. define software architecture as *"the structure(s) of the system, which comprise(s) software elements, the externally visible properties of those elements, and the relationships among them"* [Bass 2004, p.21]. Software architecture is further discussed in Chapter 2.4 in this thesis. Investigating perceived architectural risks among actual software architects will allow identification of relevant problems and provide a basis for proposing systematic solutions towards handling these problems. We hence define ***RQ4*** *as:*

What are the perceived architectural risks of CBSE-driven software evolution, and how can these risks be mitigated?

These four research questions express the overall goals of this thesis, and are further presented and motivated in Chapter 3.

Furthermore, empirical software engineering, which focuses on analyzing actual data (to validate results and propose new or revised models/abstractions), has become an important research arena. The aim of empirical software engineering is thus to move the field from merely analyzing and validating theoretical concepts towards a more scientific-based field [Tichy 1998] [Zelkowitz 1998] [Glass 2004] [Dybå 2005] [Wong 2008] [Basili 2008] [Kampenes 2009].

This research applies a combined set of methods, using qualitative methods to follow up studies performed quantitatively, as well as quantitative methods to follow up studies performed qualitatively (i.e. to explore related information in more detail).

The chosen study method will influence the collection and subsequent analysis of relevant data, as well as the eventual considerations regarding validity and generalization. In general terms, we can say that

- quantitative studies (e.g. experiments) are more concerned with the "what" and the "when" of an occurring trend, while

- qualitative studies (e.g. surveys) are more concerned with "why" this trend appears the way it is.

Empirical studies in software engineering are commonly performed through qualitative studies, quantitative studies, or a combination of these two. Combining the two types of studies provides the benefits of both, allowing them to complement each other in terms of study results towards a given study goal.

Our research is set in **three major phases**, numbered 1 – 3 (Figure 1), and combines quantitative and qualitative aspects. These phases were performed with some activities overlapping in time.

In **phase 1** (comprising two studies), we investigated the state of practices and issues of modern trends within CBSE on the development process for reusable in-house (article **P1**) and COTS/OSS (article **P2**) reusable components. The studies in this phase are comprised of qualitative surveys, combined with semi-structured interviews, of developers. We then combined the results from these two studies to answer **RQ1.**

In **phase 2**, we investigated software evolution specifically for individual reusable components on the basis of defect density (defined as the number of Trouble Reports (TRs) divided by the number of non-commented Source Lines of Code (NSLOC)) and change density (defined as the number of Change Requests (CRs) divided by NSLOC) (article **P3**). The research method used in this phase is quantitative case study, and it is geared towards answering **RQ2.**

The two aforementioned phases provided input in terms of general background, results and experience for **phase 3**, which totals three studies. Here, we investigated Test Driven Development in the context of software evolution (article **P4**), contributing to answering **RQ3**. We also investigated perceived risk and risk management aspects of software architecture evolution, based on qualitative surveys among software architects in the Norwegian software industry (articles **P5** and **P6**). These latter two studies provide results towards answering **RQ4.**

Figure 1 below shows the different phases of this research, indicating selected articles (Px, full articles in Appendix A), contributions (Cx) and secondary articles (SPx, abstracts in Appendix B).

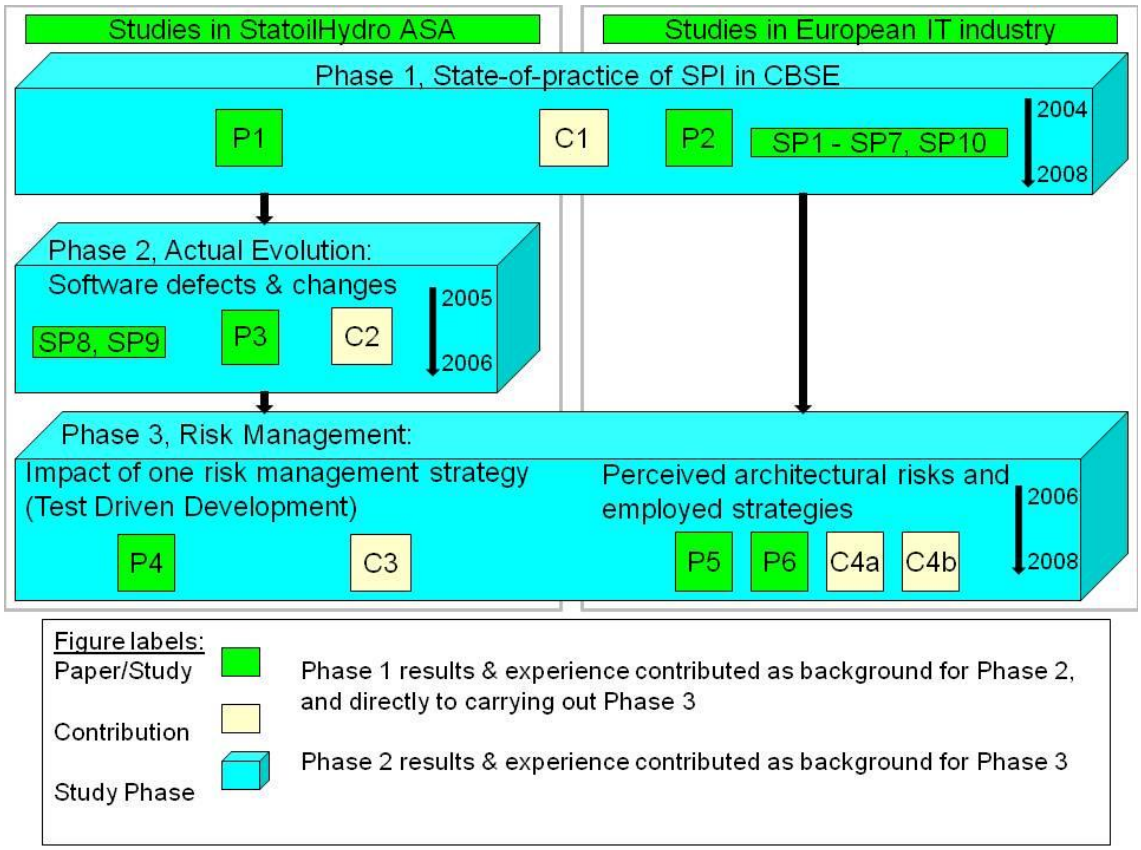

Figure 1: Phases, contributions and papers

Due to the availability of data sources, there is some overlap between the phases (as indicated by the timelines for each phase shown in Figure 1). A further outline of our work in terms of papers/studies vs. focus, context, research questions and contributions can be found in Table 1. Papers P1, P2 and P3 contribute to SEVO goal G1; papers P4, P5, and P6 contribute to SEVO goal G2; while all of the studies contribute to SEVO goals G3 and G4. We have used a combination of quantitative and qualitative methods.

Table 1: Research overview

Paper/ Study	Focus	Context	Direct Study Objects	Research Question	Contri-butions	Research Method
P1	Developers' attitudes to software reuse	Reuse-based development, IT-department of a large Oil and Gas company (StatoilHydro)	Software developers	RQ1	C1	Survey
P2	COTS/OSS impact on development processes	Software industry in Norway, Italy and Germany	Software developers	RQ1	C1	Survey
P3	Defect density and change density in CBSE-driven software evolution	Reuse-based development, IT-department of a large Oil and Gas company (StatoilHydro)	Reusable framework of components	RQ2	C2	Case study
P4	Test Driven Development and software evolution	Reuse-based development, IT-department of a large Oil and Gas company (StatoilHydro)	Reusable framework of components	RQ3	C3	Case study
P5	Perceived risks in software architecture evolution	Norwegian software industry, perceived architectural evolution	Software architects	RQ4	C4a	Survey
P6	Software architecture risk management	Norwegian software industry, perceived architectural evolution	Software architects	RQ4	C4a, C4b	Survey

The following section presents a brief outline of the papers included in this thesis, along with related contributions.

1.4 Selected Articles (P1 – P6)

The selected papers are listed below, together with an outline of my own contributions:

[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 ACM-IEEE Int'l Symposium on Empirical Software Engineering (ISESE'06), Rio de Janeiro, 21-22 September 2006, IEEE CS Press, ISBN 1-59593-218-6, pp. 242-251.

Relevance: This article encompasses a study of attitudes towards software reuse among developers, allowing us to attain more detailed information on benefits of reuse as well as success factors for software reuse, all in a CBSE context. This article contributes results from in-house development about the impact on the development process for reusable components **(RQ1 and C1).**

Contribution: I was one of the main contributors towards the study design, execution and data collection, as well as the analysis and writing of the article. I was the leading author of this article, where the work tasks were mainly divided between me and Anita Gupta. We furthermore worked on the tasks individually, while reviewing each other's work and providing major and minor comments towards the completed article. The remaining co-authors gave us their feedback towards the finalized article submitted for publication.

[P2] Jingyue Li, Reidar Conradi, Christian Bunse, Marco Torchiano, **Odd Petter N. Slyngstad,** and Maurizio Morisio: "Development with Off-The-Shelf Components: 10 Facts", In *IEEE Software*, 26(2):80-87, March/April 2009.

Relevance: This article summarizes our results and experiences from a large survey of COTS/OSS development in Europe. To this thesis, this article contributes results about impact on the development process for COTS/OSS components **(RQ1 and C1).**

Contribution: I contributed towards the research design, data collection and analysis, as well as the writing of articles resulting from this study on COTS/OSS development, which also encompasses a majority of the secondary papers (SP1-SP7, SP10) the abstracts of which are included in this thesis.

[P3] Odd Petter N. Slyngstad, Anita Gupta, Reidar Conradi, Parastoo Mohagheghi, Thea Christine Steen, and Mari Haug: "Preliminary results from an investigation of software evolution in industry", In Tom Mens, Maja D'Hondt, and Laurence Duchien (Eds.): Proc. ERCIM Workshop on Software Evolution, 6-7 April 2006, Lille, France, pp. 187-193.

Relevance: This article presents the results from an investigation on defect and change densities for individual components in a framework of reusable components. This study contributes towards our investigation of software evolution impact on individual reusable components **(RQ2 and C2).**

Contribution: I was one of the main contributors to this study, and was the leading author of this article. In summary, I contributed 50% of the work in terms of research design, data collection, analysis, and the writing of this article. The work towards writing the article was divided between me and Anita Gupta, then jointly reviewed by all the authors (who provided major and minor comments) as the individual parts were completed.

[P4] Odd Petter N. Slyngstad, Jingyue Li, Reidar Conradi, Harald Rønneberg, Einar Landre, Harald Wesenberg: "An Empirical Study of Test Driven Development in the evolution of a framework of reusable components", In Herwig Mannaert, Tadashi Ohta, Cosmin Dini, and Robert Pellerin (Eds.): Proc. The Third International Conference on Software Engineering Advances (ICSEA'08), 26-31 October 2008, Sliema, Malta, IEEE CS Press, ISBN 978-1-4244-3218-9, pp. 214-223.

Relevance: In this article, we investigate the relation between defect and change densities for traditional development versus Test Driven Development. The findings contribute towards a more detailed understanding of the impact and effectiveness of TDD for the development of reusable components in an industrial setting **(RQ3 and C3).**

Contribution: I was the main contributor (80%) to this study with regards to research design, data collection and analysis, and the writing of this article. Therefore, I was the leading author of this article, while the co-authors reviewed the work underway, giving useful comments and feedback.

[P5] Odd Petter N. Slyngstad, Jingyue Li, Reidar Conradi, M. Ali Babar: "Identifying and Understanding Architectural Risks in Software Evolution: An Empirical Study", In A. Jedlitschka and O. Salo (Eds.): Proc. 9th International Conference on Product Focused Software Development and Process Improvement (PROFES'2008), 23-25 June 2008, Frascati - Monteporzio Catone, Rome, Italy. Springer Verlag LNCS 5089, 448 pages, pp. 400-414.

Relevance: This article explores perceived risks and corresponding risk management strategies encountered and employed by actual software architects in the Norwegian software industry. The contribution towards this thesis is an investigation of risks and corresponding risk management strategies in software architecture evolution **(RQ4 and C4a).**

Contribution: I led the design of this study, and also performed the data collection and the main work of the analysis, mainly receiving useful input from Dr. Jingyue Li and my PhD advisor Dr. Reidar Conradi. I was also the leading author of this article, while the co-authors gave major and minor comments on the final article.

[P6] Odd Petter N. Slyngstad, Jingyue Li, Reidar Conradi, M. Ali Babar, Viktor Clerc, Hans van Vliet: "Risks and Risk Management in Software Architecture Evolution: An Industrial Study", In Huimin Lin, Wenhui Zhang and Shamsul Sahibuddin

(Eds.): Proc. 15th Asia-Pacific Software Engineering Conference (APSEC'08), 3-5 December 2008, Beijing, P.R. China, IEEE CS Press, pp. 101-108.

Relevance: This article further explores risks and risk management strategies relevant for the evolution of software architecture, based in part on our findings in [P5]. The contribution to this thesis is in terms of an expanded set of perceived risks and corresponding risk management strategies, towards an adapted operational matrix for risk management in software architecture evolution **(RQ4, C4a and C4b).**

Contribution: I led the design of this study, and also performed the data collection and the main work of the analysis. I also received many useful inputs from the co-authors during the course of the study. I was the leading author of this article, while the co-authors gave major and minor comments on the individual parts as they were completed, including a review round of the finalized article.

1.5 Secondary Articles (SP1 – SP10)

The remaining articles presented below were also published during the course of the thesis work. They further add background information and scope towards this thesis. The articles are listed here with a brief discussion of their outlines and my contributions, since they are not directly part of this thesis. The abstracts of these articles are included in appendix B of this thesis.

[SP1] Jingyue Li, Reidar Conradi, **Odd Petter N. Slyngstad,** Marco Torchiano, Maurizio Morisio, and Christian Bunse: "Preliminary Results from a State-of-Practice Survey on Risk Management in Off-The-Shelf Component-Based Development", In Xavier Franch and Daniel Port (Eds.): Proc. 4th International Conference on Component-Based Software Systems (ICCBSS'05), 7-11 February 2005, Bilbao, Spain, Springer LNCS 3412, pp. 278-288.

Outline/Contribution: This is the first presentation of results from a large industrial survey on risk management for COTS/OSS components. I participated in the research design, data collection and analysis here, and contributed towards 30% of the work.

[SP2] Jingyue Li, Reidar Conradi, **Odd Petter N. Slyngstad,** Christian Bunse, Umair Khan, Maurizio Morisio, and Marco Torchiano: "Barriers to Disseminating Off-The-Shelf Based Development Theories to IT Industry", position paper, In Abdallah Mohamed, Guenther Ruhe, and Armin Eberlein (Eds.): Proc. the International Workshop on Models and Processes for the Evaluation of COTS Components (MPEC'05), 21 May 2005, 4 p, ACM Press. Arranged in co-location with ICSE'05, St Louis, Missouri, USA, 15-19 May 2005.

Outline/Contribution: This article entails a follow-up study on risk management issues related to COTS/OSS components. I contributed 40% of the research design and data collection for this article.

[SP3] Jingyue Li, Reidar Conradi, **Odd Petter N. Slyngstad,** Christian Bunse, Umair Khan, Marco Torchiano, and Maurizio Morisio: "An Empirical Study on Off-the-Shelf Component Usage in Industrial Projects", In Frank Bomarius and Seija Komi-Sirviö (Eds.): Proc. the 6th International Conference on Product Focused Software Process Improvement (PROFES'05), 13-16 June 2005, Oulu, Finland, pp. 54-68, Springer LNCS 3547.

Outline/Contribution: This article describes an empirical study on the specific reasons for choosing to use either COTS or OSS components, respectively. My contribution here was towards 30% of the total work.

[SP4] Jingyue Li, Reidar Conradi, **Odd Petter N. Slyngstad**, Christian Bunse, Umair Khan, Marco Torchiano, and Maurizio Morisio: "Validation of New Theses on OTS-Based Development", In Filippo Lanubile and Carolyn Seaman (Eds.): Proc. the 11th IEEE International Software Metrics Symposium (Metrics'05), Como, Italy, 19-22 Sept. 2005, IEEE CS press, pp. 26-26 (abstract), 10 p.

Outline/Contribution: This article entails a further report on the results from a large industrial survey on risk and risk management issues for COTS/OSS-based development. I participated in questionnaire design and setup, as well as data collection in Norway, and contributed towards 40% of the total work.

[SP5] Reidar Conradi, Jingyue Li, **Odd Petter N. Slyngstad,** Vigdis By Kampenes, Christian Bunse, Maurizio Morisio, and Marco Torchiano: "Reflections on conducting an international survey of Software Engineering", In June Verner and Guilherme H. Travassos (Eds.): Proc. the International Symposium on Empirical Software Engineering (ISESE'05), Noosa Heads (Brisbane), Australia, 17-18 November 2005, IEEE CS Press, pp. 214-223.

Outline/Contribution: This article reports on the challenges, approaches, and experiences we encountered during our planning, execution, analysis and aggregation of results in the large industrial survey on risk management issues related to COTS/OSS components. My contribution here was towards 30% of the total work.

[SP6] Jingyue Li, Reidar Conradi, **Odd Petter N. Slyngstad,** Christian Bunse, Marco Torchiano, and Maurizio Morisio: "An Empirical Study on the Decision Making Process in Off-The-Shelf Component Based Development", In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa (Eds.): Proc. the Emerging Results track at the 28th International Conference on Software Engineering (ICSE 2006), 20-28 May 2006, Shanghai, P.R. China, ACM Press, pp. 897-900.

Outline/Contribution: This study is on the commonalities and differences in integrating COTS and OSS components in new development, from the large industrial survey on risk management issues related to such components. My contribution was towards 30% of the total work here.

[SP7] Jingyue Li, Marco Torchiano, Reidar Conradi, **Odd Petter N. Slyngstad,** and Christian Bunse: "A State-of-the-Practice Survey of Off-the-Shelf Component-Based Development Processes", In Maurizio Morisio (Ed.): Proc. 9th International

Conference on Software Reuse (ICSR'06), Torino, 12-15 June 2006, Springer LNCS 4039, pp. 16-28.

Outline/Contribution: This article entails a more thorough report of the larger study on risk management issues for COTS/OSS components. I participated in the research design, data collection and analysis here, and contributed towards 30% of the work.

[SP8] 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", Proc. the International Conference on Software Engineering Advances (ICSEA 2006), 29 October – 3 November 2006, Tahiti, French Polynesia, IEEE CS Press, 7 p. (due to conference format requirements). Republished in Arne Løkketangen et al. (Eds.): Proc. Norwegian Informatics Conference (NIK'06), 20-22 November 2006, Molde, Norge, Tapir Akademisk Forlag.

Outline/Contribution: This article explored the relation between component size and change profiles for reusable vs. non-reusable components. We also investigated the change type distribution (perfective, adaptive, preventive and corrective). This article entails a contrast and comparison study of reusable vs. non-reusable components. I participated in the data collection, as well as the analysis for this study. I was the second author of this article, and contributed towards approximately 50% of the work required.

[SP9] Anita Gupta, **Odd Petter N. Slyngstad,** Reidar Conradi, Parastoo Mohagheghi, Harald Rønneberg, and Einar Landre: "An Empirical Study of Defect-Density and Change-Density and their Progress over Time in Statoil ASA", In René L. Krikhaar, Chris Verhoef, and Giuseppe A. Di Lucca (Eds.): Proc. the 11th European Conference on Software Maintenance and Reengineering, Software Evolution in Complex Software Intensive Systems (CSMR 2007), 21-23 March 2007, Amsterdam, The Netherlands, IEEE Computer Society 2007, pp. 7-16.

Outline/Contribution: This article offers a more detailed view of the contrast between reusable and non-reusable components in terms of defect density and change density. It provides a quantitative view of some of the benefits of software evolution management through software reuse. My main contributions in this study were towards the analysis of the data. I also participated in the data collection, and I was the second author of this article. I performed approximately 50% of the work required for this article.

[SP10] Jingyue Li, Reidar Conradi, Christian Bunse, Marco Torchiano, **Odd Petter N. Slyngstad,** and Maurizio Morisio: "A State-of-the-Practice Survey on Risk Management in Development with Off-The-Shelf Software Components", *IEEE Transactions on Software Engineering (TSE)*, 34(2):271-286 (Feb. 2008).

Outline/Contribution: This article entails a more detailed analysis and discussion of the international industrial survey on risk and risk management for COTS/OSS components development. I participated in questionnaire design and setup, as well

as data collection in Norway, and contributed towards 40% of the total work for this survey.

1.6 Contributions of this thesis

This section summarizes the contributions of this thesis, which are discussed in more detail in Chapters 5.1 – 5.4.

C1. Improved knowledge of modern trends in CBSE and their impacts on software development processes (RQ1, articles P1 and P2). We investigated how the development process for in-house and COTS/OSS-based development is impacted through modern trends in CBSE. This yielded a detailed identification of impacts, explaining the similarities and differences for in-house reusable and COTS/OSS components. From our results, these impacts include:

- Impact for in-house reusable components (P1):
    - A defined / standardized architecture is seen as key
    - Training and knowledge sharing remain important

- Impact for external COTS/OSS components (P2):
    - Traditional processes, enriched with OTS-specific activities, can be and are being used to select and integrate OTS components

The identification of these impacts is important in order to set the stage for improvements in the processes used towards developing software.

C2. Improved understanding of evolution impact on individual reusable components in terms of defect and change densities (RQ2, article P3). Through a study of the evolution of individual reusable components, characteristics in terms of defect and change densities were elicited. We hereby gained a more detailed understanding of software evolution impact on individual reusable components.

C3. Improved understanding of the impact and effectiveness of TDD (RQ3, article P4) By carrying out a quantitative investigation on TDD, versus traditional test-last development approaches, we gained insight into impacts of software evolution for this process improvement approach.

C4a. Identification of perceived risks and related mitigation strategies specifically for the evolution of software architecture (RQ4, articles P5 and P6). Related studies have discovered that formal, documented methods are not commonly used to evaluate software architectures in industry. Moreover, the existing methods are geared towards architectural design, not evolution. We therefore performed a qualitative study of industrial software architects. A set of risks and corresponding risk management strategies were identified from this study.

C4b. An adapted operational matrix as a tool to support risk management in software architecture evolution (RQ4, article P6). We have developed an adapted operational matrix to aid risk mitigation in software architecture evolution, exploring the results from contribution **C4a**. This adapted operational matrix represents a first step towards structured mitigation of perceived risks in software architecture evolution.

1.7 Thesis Structure

The following paragraphs outline the remaining chapters of this thesis:

Chapter 2 State-of-the-art: Here we discuss the current state of the software engineering field, with focus on Component-Based Software Engineering, Software Process Improvement, as well as Software Reuse and Evolution. Furthermore, we discuss Software Architecture and Software Risk Management in more detail, as they are of particular interest for this thesis. The chapter concludes with a summary of research methods in software engineering along with their suitability towards our studies.

Chapter 3 Research Questions, Design and Implementation: We discuss our research focus and the motivation behind our research questions here. The design and implementation of our research process is also discussed in this chapter.

Chapter 4 Results: This chapter presents our results from the individual articles P1-P6, bringing them together towards the contributions of this thesis.

Chapter 5 Evaluation and Discussion: Our findings are evaluated and discussed with focus on our contributions, and also in relation to the state-of-the-art. Additionally, we have included recommendations to practitioners and a discussion of how our results fit with the overall SEVO research project goals. This chapter also contains reflections on the research context, and general recommendations to practitioners, as well as general and specific considerations regarding the validity of the contributions of this thesis.

Chapter 6 Conclusion: The main findings of this thesis are revisited, and further specific recommendations for researchers and practitioners in CBSE are outlined. Directions for possible future work are also identified in this chapter.

Appendix A: This thesis is based upon six articles (P1-P6) that have been published. These are presented in their entirety in this chapter.

Appendix B: This includes the abstracts of the ten secondary articles (SP1-SP10) in this thesis.

2 State-of-the-art

We here provide a general description of the scientific context surrounding the work in this thesis. This includes definitions of the central research areas in software engineering, and their challenges and opportunities. We also provide a more detailed description of prior work in software reuse, CBSE, software evolution, software architecture, and software risk management, as these are areas that pertain more closely to our research focus. The chapter concludes with a tally of the research methods we have applied, as well as their strengths and weaknesses.

2.1 Software Engineering

According to Finkelstein et al. [Finkelstein 2000], system engineering encompasses *software engineering* as a sub-area, together with hardware and mechanical engineering. Also along the same lines, the IEEE Standard Glossary of Software Engineering Terminology gives the following definition [IEEE 90a, p. 67]:

> *Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software*.

While system engineering refers to the overall task of developing systems, Software Engineering specifically has to do with enabling construction of large, complex and long-lived software systems with ever-changing requirements and surrounding infrastructure (executable platforms). As such, it entails important inherent implications on the processes, methods and tools (i.e. relevant technologies) surrounding the development of these larger and more complex systems.

In particular, software development incurs problems not otherwise seen in production organizations: According to Griss et al. [Griss 1993], these problems and challenges surrounding the development and maintenance of software were first described as "the software crisis" in 1968. The term 'software engineering' was first used at a NATO conference in Garmisch-Partenkirchen [Naur 1968]. Commonly referenced reports still point towards these problems and challenges resulting in frequent cancellations, gross budget and/or schedule overruns, missing or wrong functionality, or other serious deficiencies in software projects [CACM's Inside Risks 2010].

van Vliet [vanVliet 2008, p.6-8, adapted] outlines eight characteristics of software engineering:

- *Software engineering concerns the development of large programs – present-day software development projects result in systems containing a large number of (interrelated) programs or components.*
- *The central theme is mastering complexity (related to the vast number of details rather than the intrinsic complexity of the software problem) – by splitting the given software problem into smaller, more manageable pieces.*
- *Software evolves – as reality evolves. This evolution has to be kept in mind during development.*
- *The efficiency with which software is developed is of crucial importance – software projects have a high total cost and development time (this also holds for maintenance). Enabling better and more efficient methods and tools for the development and maintenance of software, especially those enabling the use and reuse of components, is an important theme within software engineering.*
- *Regular cooperation between people is an integral part of programming-in-the-large – relying on clear work distribution, communication, responsibilities, as well as enforcement e.g. through standards and procedures.*
- *The software has to support its users effectively – through fulfilling the right quality and functional requirements.*
- *Software engineering is a field in which members of one culture create artifacts on behalf of members of another culture – software engineers have to rely on knowledge of domain experts from non-software fields.*
- *Software engineering is a balancing act – most requirements are negotiable, while numerous business, technical and political constraints may influence a software development project.*

In particular, *Component Based Software Engineering* focuses on development of components with reuse. Components can be considered to be units of composition (e.g. a class library or a package in some programming language), specified such that their interfaces are separate from their implementation [Crnkovic 2002]. These components are, per definition, of independent development and deployment. This allows – ideally – improvement and change of individual components, without requiring changes in the client software (e.g. other components) that use such components, assuming the interfaces remain stable. CBSE emphasizes reuse with components, in connection with a given architectural framework and with code-level interoperability [vanVliet 2008]. The ultimate aim of CBSE is hence to provide and integrate software components that function very much like plug-in hardware components.

Research Challenges in Software Engineering

Based on the above characteristics of software engineering, some current challenges include:

- Exploiting new software processes and value chains due to COTS and OSS.
- Validating the applicability of agile methods (XP, Scrum) with respect to software architecture, quality and maintenance concerns.

- Improving the balance between rigor and relevance in empirical studies [Glass 1994].
- More systematic collection and synthesis of empirical evidence world-wide [Basili 2008].
- Combining existing and revised software methods with respect to "global" software development, i.e. supporting distributed teams.

More specifically, [Sommerville 2010] further outlines increasing diversity, reduction of time-to-market while maintaining quality, and provable trustworthy computing as key challenges. Software process improvement is inherently related to challenges in software engineering.

2.2 Software Quality and Process Improvement

The IEEE Standard Glossary of Software Engineering Terminology [IEEE 90a], focusing on a satisfied customer, defines software quality as follows:

> *The degree to which a system, component, or process meets specified requirements.*

The software-intensive industry is, and always has been, looking for ways to develop software faster, cheaper, more predictably, with requested functionality and quality (e.g. correctness and reliability), and with sufficient maintainability. Central to making "better" software-intensive systems is improving the work process(es) for developing and maintaining the appropriate software. That is, only if there is a well-defined and well-suited software process in current use can the quality of a software product become properly established [Sommerville 2010].

> *Software Process Improvement (SPI) encompasses the understanding and changing of existing processes to improve software product quality, as well as to reduce costs and development time [Sommerville 2010, p. 706, adapted]*.

According to Aaen [Aaen 2001], SPI is one of the most commonly used ways to achieve these benefits. Sommerville [Sommerville 2010, p. 710-721, adapted] outlines the following stages for SPI, in a cyclical process:

- *Process measurement – Measure current project or product, aiming for improvement of measures according to set goals. Also forms a baseline for measuring improvement effectiveness.*
    - *The time taken for a particular process to be completed – e.g. total process time, calendar time, time per resource.*
    - *The resources required for a particular process – e.g. total effort in person-months and other costs such as travel and computer resources.*
    - *The number of occurrences of a particular event – monitoring events, e.g. number of defects, number of requirements changes, average lines of code changed.*

- *Process analysis – Assessing the current process for weaknesses and bottlenecks. Development of process models.*
    - *Questionnaires and interviews – questioning engineers and managers to obtain relevant information.*
    - *Ethnographic studies – observing process resources in their work environment (more aimed at revealing subtleties and complexities not discovered through questionnaires and interviews).*
- *Process change – Proposing and introducing process changes to address identified weaknesses and bottlenecks. Cycle continues with Process measurement to determine the effectiveness of the changes.*
    - *Improvement identification – identifying ways to tackle weaknesses and bottlenecks identified in analysis.*
    - *Improvement prioritization – assessing e.g. importance, cost, and potential impact of possible process changes.*
    - *Process change introduction – integrating new process changes (e.g. procedures, methods and tools) with existing processes, allowing sufficient time to ensure compatibility.*
    - *Process training – introducing training to ensure full benefits of the process changes can be gained.*
    - *Change tuning – full effectiveness of the changes introduced can only be ensured after adjusting for minor problems which may arise once the new changes are in place.*

Management commitment at all levels is also important throughout SPI. This latter point of change tuning is also emphasized to best be an ongoing and continuous activity by Fuggetta [Fuggetta 2000].

Similar to the key stages outlined by Sommerville above, Zahran [Zahran 1998, p. 68-69, adapted] proposed a combined set of methods for SPI. This includes:

- *a software process infrastructure, i.e. organization, management and technical,*
- *a software process improvement roadmap, i.e. a model specification (such as a software capability maturity model [CMM] , ISO/IEC 15504 [ISO 2010], SPICE [SPICE] or a SWOT analysis), or results from company-specific gap analysis of current vs. future state,*
- *a software process assessment method, i.e. for assessing e.g. current process and methods, and*
- *a software process improvement plan to map assessment findings towards specific SPI initiatives.*

In SPI, as in other areas of software engineering, there is a need for empirical evaluation of proposed technologies, as well as a proper verification of potential benefits [Glass 1999]. This is because there is currently too little empirical evidence to verify these benefits, and the claims of benefits from introducing new technologies often do not match reality (i.e. currently available empirical evidence) in SPI [Glass 1999].

An example of software process improvement is Test Driven Development, a practice related to agile software development that focuses on writing unit tests prior to the actual code [George 2004]. The process of writing formalized tests for the smallest functionality increments, and then developing that functionality, is performed in a

cyclical manner until the system is built [Erdogmus 2005]. TDD tests focus on low-level unit-like testing, rather than cross-cutting or combining testing concerns. Additionally, the roles of test writer and software developer are commonly filled by the same person.

Using TDD has several *advantages*, as follows:

- TDD aids in *code comprehension* since developers explain their code through test cases and code itself, rather than more formal documentation [George 2004].
- TDD makes determining the problem source when encountering new defects more *efficient* (i.e. during development) [George 2004].
- The test cases developed using TDD comprise important assets towards further *testing* as well as the identification of newly found defects (as noted above) [George 2004].
- Software maintenance and debugging in traditional test-last development is commonly considered a low-cost activity where the code is patched, but the design and the specifications are neither examined nor changed accordingly [Hamlet 2001]. These small code changes can be up to 40 times more likely to cause further errors, meaning that new faults are commonly injected during debugging and maintenance. As TDD encourages the inclusion of new test cases to counter newly found defects, it can *reduce defect injection* caused by e.g. code maintenance.

Some *disadvantages* can also be seen in using TDD:

- TDD commonly includes *no or little design*. This works well only for well-written and well-understood code, and enables the possibility of lacking conceptual integrity. This means that when defects are found, there is no "backup" in terms of formal design and documentation [vanDeursen 2001a]. One may then miss the "big" picture [Foote 1997][Perry 1992], and thereby incur problems related to the architecture.
- The amount of effort used in writing test cases is considerable, and may be *context-dependent* [George 2004].
- *Refactoring* is used extensively to manage complexity when utilizing TDD [George 2004].
- A *high level of experience* and knowledge is needed in order to develop and maintain the test assets in TDD [vanDeursen 2001a] [vanDeursen 2001b].

Research Challenges in Software Quality and Process Improvement

- Aaen et al. [Aaen 2001] comment that although SPI has received much attention in later years, there is still the need for additional focus on specific sub-areas within SPI. Classifying SPI approaches according to management methods, improvement approaches, and perspectives, they outline underlying issues as becoming the more important challenges towards future efforts in SPI research.

- Challenging SPI issues [Conradi 2002] include context, commitment, and the actual improvement process, as well as the establishment of a knowledge base on experiences with SPI.
- According to Hansen et al. [Hansen 2004], SPI research is still too focused on describing possible improvement approaches, rather than evaluating these in practice and learning from the experience gained. Having reviewed 322 published articles on SPI, they conclude that more practical evaluation is needed to draw valuable conclusions, and that the research thus far has been centered on the Capability Maturity Model [CMM].
- More specific challenges thus include further empirical studies on methods and models for SPI, as well as building a knowledge base of combined best practices. An example here would be TDD, as discussed above.

2.3 Software Reuse (in-house software, COTS, and OSS) and CBSE

Software reuse relies on CBSE to provide a potent way to manage functionality, quality, schedule, and cost issues in software development. The CBSE research area has a history since 1968 [McIlroy 1968] [Sommerville 2010], focusing on enabling common/shared components for in-house reuse between projects or departments of an organization.

> *The modern view does not restrict the notion of software reuse to component reuse. Design information can be reused also, as can other forms of knowledge gathered during software construction [vanVliet 2008, p.573]*

In relation to the software lifecycle, software reuse can be divided into two main process models; *development for reuse* and *development with reuse* [vanVliet 2008]. The former refers to the development and generalization (often by reengineering) of components specifically for future reuse. The latter means incorporating such reusable components when developing new software systems.

In CBSE, development can be performed using components from in-house, or provided by third parties. Third party software components are commonly categorized as either Commercial Off-The-Shelf software (i.e. commercial and often closed source, but 1/3 actually come with the source [Li 2004]), or Open Source Software components (where the source is more or less freely available).

Johnson and Foote [Johnson 1998] claim that abstractions that are useful towards future reuse are "discovered rather than designed". This means that components developed for standardized reuse are commonly based on already acquired components or insights (often by reengineering), rather than being designed and implemented from scratch. However, there is little empirical evidence to support this and similar claims [Jacobson 1997]. Furthermore, reuse projects can incur additional reengineering or refactoring costs (up to twice the original development cost), in order to prepare for later, assumed reuse. A common break-even point seems to occur after one to two cases of reuse [Lim 1994]. Also, effective reuse through CBSE assumes a relatively stable system architecture and application domain.

As an example of development for reuse, an overview by Basili and Boehm [Basili 2001] shows that COTS components (which are meant for use in new development) appear to be released on a three-quarter yearly basis, with backward compatibility only guaranteed for the three latest releases of a given component. Again, the empirical evidence base is weak, as there are few validated and efficient methods to integrate components in new development, or to analyze the consequences and benefits of using such components [Boehm 1998] [Morisio 2000].

Reuse entails six main dimensions, according to van Vliet [vanVliet 2008, p. 574]:

- ***Substance:*** *the reuse of components, concepts and procedures.*
- ***Scope:*** *the reuse of generic components across domains, or specific components within a particular domain.*
- ***Approach:*** *planned, systematic or ad-hoc, opportunistic reuse.*
- ***Technique:*** *composing systems from existing reusable components, or reusing domain knowledge to generate components (e.g. with application generators).*
- ***Usage:*** *components reused "as is", or modified slightly to fit new applications.*
- ***Resulting Product:*** *e.g the source code, object, design, architecture, knowledge.*

When it comes to possible differences in reuse dimensions, Li et al. [Li 2004] found that the issues were the same when using components built in-house and when using COTS. Also, their results indicated that special system repositories or portals do not contribute much towards successful software reuse, as also indicated by [Frakes 1995] [Morisio 2000]. An empirical evaluation of the potential impact of software reuse was shown in Mohagheghi and Conradi [Mohagheghi & Conradi 2004b] [Mohagheghi & Conradi 2008]. As aforementioned, they found that reused components generally have lower defect density and lower code-change density than non-reused ones. They also indicated that industrial reuse has implicit advantages that can improve software quality. These advantages include reusable components' independence of a specific programming language.

Software testing has moved from mere error discovery towards prevention [Bertolino 2007]. While CBSE components need to be retested when being reused, their interfaces are defined by component models whose information is insufficient for functional testing. To remedy this, co-packaging with additional information, built-in testing (i.e. test-cases), and verification means have been proposed.

COTS

Commercial Off-The-Shelf software components are defined in several different ways by different authors. Oberndorf [Oberndorf 1997] defines them as existing, ready-to-use pieces of software, that are publicly available for purchase. Similarly, Vidger et al. [Vidger 1996][Vidger 1997] describe COTS components as pre-existing software, commercially produced with the vendor usually retaining control over requirements, release schedule, and evolution. Artifacts such as source code, documentation, and complete specifications, commonly remain out of reach for the customer, although, as mentioned, research reveals that 1/3 of COTS components actually come with source code [Li 2004] anyway.

Basili and Boehm [Basili 2001] outline the following three key characteristics of COTS components:

1. They're closed source, i.e. normally, no access to the source code is given.
2. Their development and evolution is controlled by the respective vendor.
3. Their installed base is individually non-trivial.

Classification of COTS software [Morisio 2002] can be based on e.g. cost models, selection methods, architectures, and testing/validation techniques. Also, using COTS components leads to reduced time-to-market and cost, due to improved productivity [Voas 1998a]. However, COTS components also present considerable risks to development projects [Voas 1998b]. These include unknown quality properties and vendor stability towards maintenance support. A thorough summary of the pros and cons of using COTS components is provided by Boehm and Basili [Boehm 1999].

OTS: COTS and OSS

Other authors blur the line between COTS and OSS components. For instance, based on an empirically modest study of 7 companies, Torchiano and Morisio [Torchiano 2004] define OTS components as either commercially available or open source, specifying that regardless of type they are usually treated as closed source. They further describe such components as procured externally by the development project, and provided independently from any operating system, development environment, or platform, but integrated into the final system. Furthermore, the acquirer controls neither their features nor their evolution.

OSS

The development and evolution of Open Source Software components is based on the ability of programmers to freely use the source code to enable adaptations and improvements of the original software [OSI 1998-2010]. Concerning OSS, the following definition is used in this thesis:

> *Open Source Software is software being developed under a license compatible with FSF [FSF 1985-2010] or OSI [OSI 1998-2010] licenses.*

Open Source Software differs slightly from "Free Software", a term coined in 1985 by Stallmann [Stallmann 2005]. The latter is meant as in free speech – focusing on abstract principles, while the former puts more emphasis on the concrete advantages and disadvantages of OSS components.

Madanmohan et al. [Madanmohan2004], Ruffin et al. [Ruffin 2004] and Fitzgerald et al. [Fitzgerald 2004] together outline the following advantages of OSS:

1. OSS is publicly available, and the parallel distribution enables faster development.
2. OSS components can be as reliable, efficient and robust as their conventional cousins, if not more so.
3. OSS holds the potential to avoid the threat of vendor instability or "lock-in" on support of maintenance and further evolution (although this problem can be more technology than cost-dependent).

4. The parallel development efforts towards OSS allow for faster updates and bug fixes.
5. No additional licensing is required towards additional installations of the same software.

Additionally, the generous cooperation in communities, which accompanies OSS, can also be advantageous [Ayala 2007]. Community-centered development of OSS components and systems can aid in incrementally populating available components, as well as in synchronizing the efforts required towards OSS component selection. This community approach also enables systematic support for component selection and evaluation. Examples of communities include the multitude of components available through portals such as http://www.SourceForge.net, which currently number more than half a million. However, establishing or attracting a community can be a major challenge (as summarized by Hauge et al. [Hauge 2009b]), involving considerable investment, effort and support within the developing organization. An example showing that industrial participation in OSS communities is rare was found by Chen et al. [Chen 2008]. Their results indicated that only 9% of investigated projects participated actively in such communities.

Nevertheless, OSS has been widely adopted by industry [Hauge 2009c], with about 50% of companies reporting that they use OSS components in their software development. Additionally, over 15% of the companies receive more than 40% of their income through providing OSS software and/or services. Adoption of OSS has several perceived benefits, such as lowered costs, attractive and future-oriented technology, and ease of use (information plus source code) [Hauge 2010]. However, adopting OSS is also not seen as risk-free. Rather, obtaining support and expertise, OSS component selection, potentially hidden costs of changes, unclear liabilities/responsibilities, and possibly uncontrolled adoption/modification due to availability are perceived problems in industry. The authors in [Hauge 2010] therefore also outline risk mitigation strategies, focusing on increasing employees' skills and awareness, ensuring top management commitment, and avoiding technology "lock-in" (whether OSS or proprietary) altogether.

Research Challenges in Software Reuse (in-house, COTS, and OSS) and CBSE

- The main challenges in future research on software reuse and CBSE deal with the specification, implementation and deployment of components. Key issues include definition inaccuracies, unclear relationships between quality requirements of system vs. component(s), and insufficient technology support within CBSE to properly specify quality properties [Crnkovic 2002].
- Bass et al. [Bass 2001, p. 25] outline the following important challenges, in order of importance:
    - *lack of available components,*
    - *lack of stable component technology standards,*
    - *lack of component certification, and*
    - *lack of quality methodologies for building component systems.*

- When developing with reusable components, the ability to match new and pre-stated requirements to a portfolio of reusable components is important. Obtaining sufficient knowledge of these components, as well as being able to reuse them with little or no modification, are paramount issues [Mili 1995]. Component understanding, validation and integration are also important challenges in in-house software reuse [Pooley 2008].
- Another issue is the impact of the increased use of COTS and OSS components, on e.g. requirements negotiation [Li 2004]. It is therefore important to re-evaluate software reuse issues from a developer's perspective. In particular, we need to investigate the possible benefits, disadvantages and advantages regarding successful reuse of software components. It is also important to involve the documentation and quality specifications of reusable components that are available to the developers, who reuse them in new development.
- Another concern in future research on software reuse and CBSE is the combination of quality attributes with respect to component reuse, as developers may not know the complete specification or dependencies of a given component. It is therefore difficult to know how these components can be supported by the integrating system [Voas 2001].
- A related issue is that, while solutions have been proposed for the technical testing of CBSE components [Bertolino 2007], theoretical testing of these components remains a challenge. That is, how can the characteristics of a completed system be determined from individual components' testing? Bertolino [Bertolino 2007] discusses the following directions for challenges related to compositional testing (i.e. how to reuse individual units', components' or subsystems' test results in combination towards determining conclusions and further testing with respect to the overall system):
    - Component-based software reliability – foundational theory [Hamlet 2006].
    - Assume-guarantee reasoning – conclude global behaviors from single component test traces [Blundell 2005].
    - Ioco-test correctness – single, parallel components vs. their integration [vanderBijl 2003].
    - Fault model and test case selection procedure for integration glue code [Gotzhein 2006].

COTS and OSS

- Ruffin et al. [Ruffin 2004] and Fitzgerald et al. [Fitzgerald 2004] describe challenges related to OSS components:
    - OSS increases the risk that low-performing developers will get involved.
    - Tasks other than plain development are often below par in OSS projects. These include documentation, testing, and maintenance support.
    - Including OSS in the source code of another product may require licensing permission – otherwise, adverse consequences include damage claims and resulting termination of e.g. support and distribution.
    - Technical support in OSS is based on community participation. The successful development of OSS software and the corresponding developer community are thus codependent [Scacchi 2006a]. Moreover,

a stable core developer team is required to secure the continuity of an OSS project [Järvensivu 2008], as it can be difficult for OSS providers to perform support through a community. Even so, a shift from the less structured email lists and bulletin board support of yesterday towards more professional, end-to-end support is seen within OSS, as customers are becoming willing to pay for such support [Fitzgerald 2006].

- Some studies have investigated the organization and management of OSS projects [Feller 2002]. Other studies have attempted to capture successful recipes for including OSS in commercial development [Madanmohan 2004], as the professionalization of OSS development (i.e. the involvement of regular developers rather than volunteers) may be a step in the right direction to mitigate inherent OSS challenges. However, in comparison with COTS components, there's yet no clear empirical evidence showing claimed advantages of OSS, such as faster development and evolution of the system [Paulson 2004].
- Furthermore, COTS and OSS development may not fit with traditional development methods and processes. Boehm and Basili [Boehm 1999] consider the waterfall and evolutionary development models unsuitable for development of COTS-based systems. Nevertheless, Li et al. [Li 2006] show that standard lifecycle methods, already being used in companies, are indeed adaptable to development of systems based on COTS components.

2.4 Software Architecture

Paramount to the continued successful maintenance and evolution of a software system, especially one relying on CBSE, is its software architecture, constituting its central structure. This structure is made up of software parts (components), detailing their externally visible properties ("interface(s)" of both in-going and out-going calls), as well as how they interrelate. A well-defined software architecture is one of the key factors in successfully developing and evolving a non-trivial system or a family of systems. It also functions as a framework for early design decisions to achieve functional and quality requirements. In addition, it has an important influence on the composition and work coordination of a software project. Poor architecture often contributes to project inefficiencies, poor communication and documentation, and inaccurate decision making. The below definition of software architecture refers to software elements, which we will interpret as components in a CBSE context [Bass 2004, p. 21]:

> Software architecture can be defined as *the structure(s) of the system, which comprise(s) software elements, the externally visible properties of those elements, and the relationships among them.*

According to van Vliet [vanVliet 2008, p. 290-291], there are three main purposes of software architecture:

- *It is an important vehicle for stakeholder communication; a description which can easily be communicated to customers etc. to highlight the main characteristics of a software system.*

- *It helps capture early design decisions; specific functionality is explicitly assigned to particular components in the architecture and also yields a basis for analysis.*
- *It constitutes a transferable software system abstraction, yielding a basis for software reuse.*

A well-documented, semi-formal system architecture description also aids in the analysis of various system qualities [Bass 2004]. The specific role of a system architect has been established to define and evolve such descriptions.

The architecture is influenced by the environment or context, including stakeholders, the developing organization, the architect's knowledge and experience, and the technical and organizational environment [vanVliet 2008]. The software architecture in turn influences its environment, e.g. by adding to the developing organization's experience base or becoming an asset for reuse. This cycle of mutual influence has been termed the "Architecture Business Cycle" [Bass 2004].

The software architecture of a non-trivial software system strongly influences its quality attributes, such as reliability, availability, modifiability, performance, testability, usability and security [Bass 2004]. Nevertheless, we should keep in mind that several taxonomies of software quality exist. A thorough discussion of these taxonomies is beyond the scope of this thesis, but can be found in [vanVliet 2008].

Considering interoperability between systems with different architectures, Service-Oriented Architectures (SOA) aim to provide for seamless operation between various entities, e.g. through Web Services, thus offering platform and language independence [Haller 2005]. Web services thus allow software systems to expose their capabilities as services for mutual use, with minimum overhead and maximum flexibility [Booth 2004].

Linking architecture and software evolution, *architectural evolution* is the result when evolutionary changes to the software (as defined for *software evolution* in Chapter 2.5) cause the architecture to be altered.

Research Challenges in Software Architecture

- Better knowledge and understanding about architectural evolution risks may help the development of improved strategies to mitigate these risks and make sure the project is delivered on budget and schedule (failure of the software architecture can permeate the entire project and cause it to fail, e.g. due to missing or incorrect architectural information [Buchgeber 2008]).
- Similarly, changes to the software architecture can cause subsequent changes in many components of a CBSE-driven software system [Bass 2004]. It is therefore imperative to be aware of the possible risks incurred on the software architecture through software evolution.
- Software architecture imposes structure and order [Bass 2004], taking the long-term perspective over many software releases. This can in some ways be seen as opposite to the short-term focus of agile development methods such as SCRUM [Schwaber & Sutherland 2010] on short e.g. 24 hour daily cycles and e.g. monthly development sprints. Uniting these two aspects to provide the benefits of both short-term and long-term perspectives is an important research challenge.

- So far, investigations on software architecture risks for CBSE-driven systems have focused on structured output from e.g. ATAM/SAAM/ALMA reports [Babar 2007b], and evolutionary aspects have not been taken specifically into account [P5]. Including these aspects (and investigating the evaluation methods actually used in industry) in future studies, is paramount towards capturing empirical data for comparative studies, as well as towards suggesting effective process improvements.

2.5 Software Maintenance and Evolution

Software maintenance is the updating performed on already released software in order to keep the system running and up-to-date. It is reported to consume upwards of 50% of the total software costs [Sommerville 2010]. Maintenance can be *corrective* (fixing defects), *preventive* (improving future maintainability e.g. by refactoring), *adaptive* (alterations related to platform or environment), or *perfective* (requirements being modified, extended or reduced, and performance enhancements). It is closely related to software evolution.

There is little agreement on a definition for software evolution in the research literature, and different views currently exist on the topic:

- One as part of the other:
  - Evolution as part of maintenance: Some researchers see it as activity that fits under the maintenance "umbrella" [Sommerville 2010].
  - Evolution as encompassing maintenance: Belady and Lehman [Belady 1976] first used the following definition of software evolution: "*….the dynamic behaviour of programming systems as they are maintained and enhanced over their lifetimes…"*
- Evolution as non-corrective (i.e. perfective, preventive, adaptive) changes, maintenance as corrective changes: Referring to the accumulated non-corrective changes on software between system versions [Mohagheghi & Conradi 2004a], as opposed to corrective changes termed as maintenance.
- Evolution as a lifecycle step: Yet others are of the opinion that *evolution* describes the part of the software lifecycle where requirements are still changing and the software is in production, following the initial release [Bennett 2000]. Software then enters *maintenance* once the ability to undertake changes without compromising the soundness of the architecture has been lost. This view represents a focus on the time aspect of changes, rather than the type.

We consider maintenance as having to do with maintaining the status quo, that is, correcting defects, while software evolution encompasses preventive, adaptive and perfective changes (as described above). Thus, in attempting to define *software evolution*, we choose to build on the view that evolution encompasses non-corrective changes; i.e. the process of improving and adapting a system's functionality and performance between releases. This process occurs through absorption of new and revised requirements from developers and users, and through adaptations to a continuously changing environment.

Software evolution, then, is *the systematic and dynamic updating in new/current development or reengineering from past development of component(s) (source code) or other artifact(s) to*

a) accommodate new functionality,
b) improve the existing functionality, or
c) enhance the performance or other quality attribute(s) of such artifact(s) between different releases [P5, p. 3].

The first systematic studies on software evolution were undertaken by Manny Lehman on the OS360 system at IBM [Lehman 1997]. An initial set of Lehman's "*laws of evolution"* were proposed, and later revised, refined and validated through the FEAST/1 [Lehman 1985] and FEAST/2 [Lehman 1995] research projects. These "laws" function as a guide to the evolutionary software development process and the construction of software tools. They describe the behavior of E-type software, defined as "software used for problem-solving or application-addressing in a real-world domain". This type of evolving software is accepted based on its quality, performance and usability, and hence cannot be proven correct [Lehman 2001]. In contrast, there is the less general S-type software, which by definition can be accepted based on satisfying its specification. In summary, most software systems undergo evolution perennially, potentially affecting all aspects of the system. This leads to changes in the design and objectives of the system, i.e. forming a feedback cycle. The change process eventually reaches a point where it is no longer feasible to maintain the given software, in terms of required resources.

Evaluations of Lehman's laws based on empirical evidence have also recently been undertaken by Mens et al. [Mens 2008] (on seven releases of Eclipse at approximately 2 million NSLOC) and Xie et al. [Xie 2009] (on 653 combined releases of seven different OSS applications, ranging from 5000 NSLOC to over 1 million NSLOC). The laws of continuous change and growth were readily supported by both of these studies. The law of increasing complexity was confirmed by [Xie 2009], while the outcomes of the evaluation for the remaining laws were found to depend on operational definitions.

A successful example of software system evolution is described in a study by Townsend [Townsend 1997]. The central factor to incorporating reuse in their study was the ability to maintain and share an enterprise-wide object model. This made it possible to access information per customer and per account.

Mockus et al. [Mockus 2000] performed a large scale, empirical software maintenance study at Lucent labs on a multi-million line telecommunications system. They investigated over one million change and error requests/reports, focusing on the type, frequency and size of changes incurred. They found component change frequencies of one change every 10 days, and an average change size of roughly 10 lines of code. Another study on defects in six releases of a large legacy software system (each of approximately 20 million NSLOC) was performed by Li et al. [Li 2009], investigating the pervasive multiple-component defects. Their findings showed that 6-8% of all defects per release are pervasive, and that over 70% of these pervasive defects were located to 20% of the studied components. Further, they found that the pervasive defects required 20-30 times the average number of changes to fix versus non-pervasive defects. Finally, more than 80% of components affected by pervasive defects in one release remained prone to this type of defect in subsequent releases.

Stability and reliability of a component is related to its change density (number of changes per NSLOC), caused by change requests and defect reports. Studies indicate that reusable components have a lower defect density than non-reusable ones [Mockus 2000] [Townsend 1997]. Additionally, reusable components have been shown to have a lower change density (being more stable) than non-reusable components [Mohagheghi & Conradi 2004b].

An alternative to NSLOC for measuring system size is to use the number of classes and methods. These metrics can then serve as a basis for determining the complexity of a software system and its components in order to provide for more fine-grained measurements [Gupta 2010], but does require access to the actual source code.

Yet another approach to measuring system size is to use Function Points [Albrecht 1979] [Umholtz 1994], where a point-based weight is assigned each method based on function type and complexity. One problem with this approach is that it tends to focus on user-oriented requirements, while hiding internal (e.g. algorithmic) functionality. Also, the number of different Function Point metrics currently exceeds 22 variants, and conversion between these is generally problematic [Jones 2008]. There is currently no standard method for counting Function Points that includes algorithmic complexity recognized by the ISO [ISO 2010].

The impact of evolution on a software project has many dimensions. These include:

- Additional costs in terms of effort, delays, and external resources,
- Incorporating lessons-learned from evolving reusable components, and
- Impact on quality attributes, such as reliability, availability, modifiability, performance, testability, usability, security, etc.

Furthermore, over the last decades major efforts have been made towards understanding the issues involved in reuse and to discover the benefits and disadvantages of different approaches within the field. A core point of software reuse by CBSE is the ability to manage software evolution through reusing "pluggable" and independent components systematically, while taking new requirements into account.

Research Challenges in Software Maintenance and Evolution

- Sommerville [Sommerville 2010, p. 428, adapted] lists the following challenges with respect to CBSE-driven software evolution management programs, focusing on software reuse:
    - *Increased maintenance costs – if the source code of the reused component or system is unavailable.*
    - *Lack of tool support – some tools may not support development with reuse.*
    - *Not-invented-here syndrome – engineers may prefer to rewrite rather than reuse, based on trust and challenge.*
    - *Creating, maintaining and using a component library – can be expensive, and development processes also have to be adapted.*
    - *Finding, understanding and adapting reusable components – engineers have to have a certain level of confidence before development process adaptation is possible.*

  More targeted process improvements towards existing development processes used in industry are needed to allow for incorporation of e.g. lessons-learned.

- The different definitions of software maintenance, in comparison with software evolution, continue to be a challenge towards systematic comparative studies. Another related challenge is the extensive duration (15-20 years) needed to properly study software evolution.
- Bennett and Rajlich [Bennett 2000] mention lost business opportunities due to the inability to change the software reliably enough to meet new requirements. Studies have explored issues related to densities or frequencies of changes and/or defects [Mockus 2000] [Mohagheghi & Conradi 2004b]. An important challenge is to research new ways to reduce the effort required for handling maintenance activities associated with the current frequency/density levels.

2.6 Software Risk Management

According to Boehm, a risk is any issue that can affect a project adversely if not handled correctly [Boehm 1991]. That is, each risk can be considered as a set of conditions (i.e. what can potentially go wrong) and consequences (i.e. how do the conditions affect a given software project).

A common trend in software engineering is to take the naïve view of either ignoring or underestimating the impact of risks, assuming success from the start without making explicit efforts towards handling potential problem issues [vanVliet 2008, p. 198-199, adapted]; According to van Vliet, a risk management strategy should entail the following steps in a cyclical manner:

- *Identify the risks.*
- *Determine the risk exposure.*
- *Develop strategies to mitigate the risks:*
  - *Avoidance through precautions.*
  - *Transfer through developing alternative solutions.*
  - *Acceptance through enabling a contingency plan.*
- *Handle risks through monitoring the risk factors and learning from the experience gained.*

Top risk factors for a software development project were identified by Boehm [Boehm 1991, p. 35] and Jones [Jones 2008, p. 415] (adapted and combined):

- *Personnel shortfalls (inexperience with domain, tools, personnel turnover etc.)*
- *Unrealistic schedule or budget estimates due to inaccurate estimation and schedule planning*
- *Incorrect functionality caused by e.g. misinterpretation of customer needs*
- *Incorrect user interface*
- *Implementing "nice" features not requested by the customer*
- *Requirements volatility/unstable requirements – requirements changes increase rework*
- *Quality or functionality problems with external components*
- *Subcontractor problems – inadequate quality in the work/skills delivered/provided*
- *Real-time performance or quality shortfalls of (parts of) the software system*
- *Straining computer science capabilities*

- *Absolute failure or cancellation*
- *Excessive schedule pressure*
- *Inadequate staff or inadequate skills*

Boehm has also presented a framework for risk management [Boehm 1991], shown in Figure 2. The first step in this framework includes risk assessment, dealing with risk identification, analysis, and prioritization. The second step in Boehm's framework encompasses risk control, dealing with risk management planning, resolution and monitoring. This second step focuses on problem mitigation, i.e. handling problems to minimize their impact.

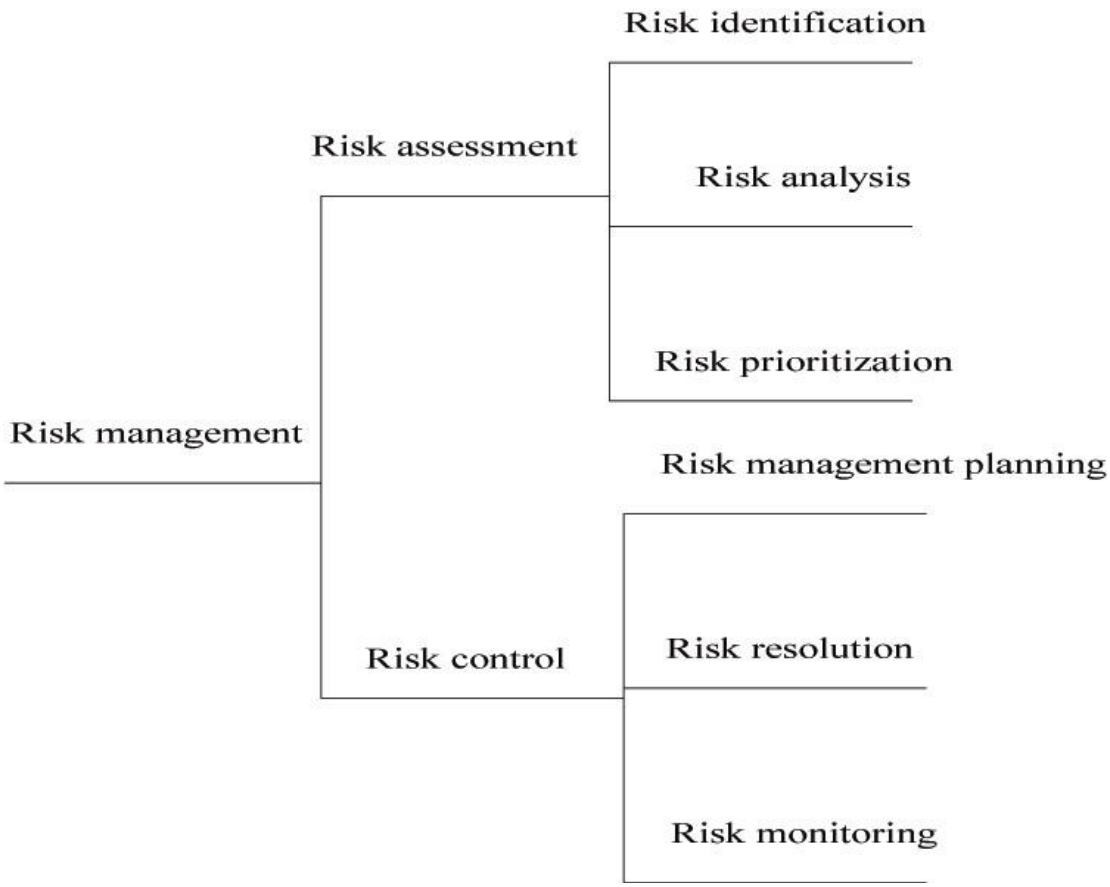

Figure 2: Overview of Boehm's framework [Boehm 1991, p. 34]

An additional risk item classification can be found in Barki et al. [Barki 1993], where the authors developed a tool for assessing project risks based on 35 risk variables. Another is the Software Engineering Institute's (SEI) Taxonomy-based tool [Carr 1993], which uses 194 questions in total to assess the risks of a project. These and other taxonomies cover large areas of software risk and risk mitigation issues. Nevertheless, there is a need for context-based risk management [Moynihan 1997].

In the research literature, risks and risk management strategies are commonly studied in relation to general software development [Boehm 1988] [Gemmer 1997] [Hecht

2004]. That is, risks are identified on the project level [Ropponen 2000] [Boehm 1991] [Keil 1998]. Similarly, software architecture studies often focus on the design, implementation and maintenance of the architecture. While these results are important as a basis for further research, there has been little effort to study risk management in the context of software architecture [Bass 2007] [O'Connell 2006].

Software architecture evaluation is widely known as an important and effective way to assess architectural risks [Bass 2004] [Babar 2007a]. In order to identify, analyze and prioritize risks [Boehm 1991], we need effective methods or mechanisms for software architecture evaluation. Such mechanisms are intended to help validate architecture design decisions with respect to required quality attributes (such as those mentioned for Software Architecture in Chapter 2.4).

We use the following definition for architectural evolution risks:

> *The issues or problems that can potentially have negative effects on the software architecture of a system as it evolves over time, hence compromising the continued success of the architecture* [P5, p. 3].

This definition is based on the research performed (on general software risk management) by Boehm [Boehm 1991], Ropponen [Ropponen 2000], and Gemmer [Gemmer 1997]. Here, continued success of the architecture refers to the ability of software architects to update the architecture description e.g. to accommodate new or altered requirements.

In this thesis, we want to obtain insight into the perceived risks and related risk management strategies in relation to software architecture evolution, as they are encountered and employed in industry. That is, we investigate the steps of risk identification, analysis and prioritization, as well as risk planning and resolution [Boehm 1991]. The issues pertaining to risk assurance or monitoring [Boehm 1991] are left for future work and not explored here.

As mentioned in Chapter 2.4, software architecture constitutes the central part of a software system [Bass 2004]. Therefore, a proper focus on the software architecture is required for the project to remain on budget and schedule. Changes to the software architecture can also cause subsequent changes in many components of a software system [Bass 2004]. It is therefore crucial to be aware of the possible risks incurred on the software architecture through software evolution.

Research Challenges in Software Risk Management

- Although several possible concepts and related activities towards effective risk management in CBSE have been proposed, there is a lack of actual empirical studies in the area [Glass 2001]. That is, the actual value and effectiveness of the proposed activities and tools remain largely unknown.
- Risk management activities (particularly related to identification and monitoring of risks) are commonly assumed to have a high cost and comparatively low return value [Odzaly 2009]. Reducing the perceived costs and improving the perceived benefits of risk management may aid towards increasing the adoption of proper risk management methods [Bannerman 2008].

- Project managers commonly lack practical techniques and tools for risk management in software processes [Liu 2009], as these are often either too general or too limited in their applicability. Developing tools and techniques that are easily combinable with existing software engineering processes and practices is an important research focus.
- Prior architecture analysis studies [Bass 2007] [O'Connell 2006] have focused on structured analysis outputs as a method to discover risks. However, the analysis methods actually used in industry are widely distributed [Babar 2007a]. Investigating a broader range of analysis methods may help to discover risk issues potentially overlooked by earlier studies.
- Proper management on the technical, process and organization level [Boehm 1988] [Gemmer 1994] [Hecht 2004] makes it possible to minimize the potentially far-reaching impacts of these risks [Boehm 1991]. In this thesis, we investigate perceived risks in, and actual mitigation strategies towards, software architecture evolution.

2.7 Research Methods in Software Engineering

We now provide a general overview of research methods pertaining to Software Engineering, outlining strengths and weaknesses of each of those used in the work in this thesis.

Empirical software engineering has risen to fulfill the need for systematic evaluation of e.g. proposed methods and tools in software engineering. Researchers in software engineering have mainly put forth new technologies [Glass 2004] i.e. a rapid development and oversell of such technologies, assuming "the sky is always blue" (i.e. new technologies will always work). Systematic empirical studies with validation of their potential benefits are grossly lacking.

Research in empirical software engineering involves applying the scientific method to consolidate knowledge in the field. This occurs by observing, reflecting and experimenting in a systematic manner [Endres 2003]. The research can be aimed towards exploring one or several (possibly unknown) parameters, describing a distribution or characteristic, or reasoning about and exploring the specific methods and applications.

There are three main types of empirical studies: quantitative, qualitative, and mixed-method [Wohlin 2000][Creswell 2003].

In quantitative studies, researchers aim to show a cause-effect relationship, verify hypotheses or test theories. This is performed through the investigation of one or more study objects in combination, while attempting to minimize contextual effects (i.e. "noise").

When it comes to qualitative studies, the purpose is rather to draw information from a natural environment or social context [Denzin 1994]. Here, the results in the given context are commonly obtained and interpreted through secondary levels, e.g. developers working with the study objects in their social environment (i.e. context is essential and not "noise").

Finally, the mixed-method approach implies that the combination of quantitative and qualitative methods complement each other with respect to individual scopes,

strengths, limitations and biases, e.g. to combine data mining with structured interviews. Also, the mixed-method approach can be used for triangulation of data [Yin 2003], performing multiple or alternating collections of data from different sources to address the same issue(s). The mixed-method approach is commonly more effective than using either quantitative or qualitative studies in isolation [Seaman 1999]. These should therefore be considered as complementary and not competitive study types. Basili et al. [Basili 1986] and Seaman et al. [Seaman 1999] discuss how to best perform this type of combination study. As the boundaries are flexible, qualitative and quantitative methods can e.g. be combined in surveys.

Empirical research strategies can also be classified into different categories, based on evaluation purpose, type of strategy (e.g. technique, method or tool), and investigation conditions. Zelkowitz et al. [Zelkowitz 1998] classified twelve technology validation models by three data collection methods (called observational, historical and controlled). These twelve classified models include project-oriented ones, such as case study and project monitoring, as well as product-oriented ones, such as static analysis and simulation. Interestingly, the following three research methods: survey [Robson 2002][Fowler 2001], action research [Davison 2004] and grounded theory [Creswell 2003] appear not to be investigated by Zelkowitz et al. [Zelkowitz 1998].

The following principal types of investigations are commonly used in software engineering:

- **Experiment** – where the study is performed in a controlled setting ("in vitro"). A randomization process is used for assigning subjects to so-called treatments, that details the tasks to be repeated by each subject on some relevant objects. One or more variables are then manipulated while controlling all the others, and the impact is measured. This then provides the basis for statistical analysis. Experiment subtypes range from true experiments (i.e. with randomized design) to quasi- (i.e. with non-randomized design) and single-subject experiments. Experiments in a university setting are more common, while industrial or professional experiments are less common. To illustrate the state of practice, a survey in software engineering showed that only 1.9% of the scientific articles published between 1993 and 2002 in the 12 leading software engineering conferences and journals were in fact on controlled experiments [Sjøberg 2005]. An example of an experiment using professionals as test subjects is a controlled experiment on the effect of a delegated vs. centralized control style on maintainability of object-oriented software [Arisholm 2004]. Here, 99 professionals from several consulting companies and 59 students took part in an experiment decentralized via the web.
- **Case study** – where the aim is to investigate the effect of some new method/activity in the development process or a new technology in a software system over a given period of time ("in vivo"). Commonly, the objective is to track or establish a relationship between particular attributes [Yin 2003], for instance defect density in Java vs. C++ software.
- **Survey** – where the data is gathered from a sample of subjects through interviews or questionnaires, which in turn are analyzed to describe and explain the observed effects. They can be e.g. cross-sectional or longitudinal in nature, and generalization is usually to the population from which the sample was taken [Robson 2002] [Fowler 2001] [Conradi 2005].

- **Action research** – where the researchers are proactively involved in the operations of and changes made to the study object during the investigation, while still performing investigations based on these operations and changes [Baskerville 1999] [Davison 2004]. In action research, research and practice are allowed to inform and influence each other continuously during the study.
- **Grounded theory** – where a general, abstract theory is extracted ("grounded") based on empirical data. Grounded theory can be characterized by a constant comparison of data to emerging categories. It also entails theoretical sampling over varying groups to maximize similarities and differences in the information [Creswell 2003]. Textual analysis is a commonly used technique in connection with grounded theory.

Empirical studies are, unfortunately, not plentiful in software engineering. Glass et al. [Glass 2004] claim in a review that a mere 14% of the published studies in Software Engineering actually evaluate some phenomenon or relationship empirically. Another review by Ramesh et al. [Ramesh 2004] found that only 11% of the considered studies applied empirical analysis. In contrast, both reviews found that the majority (70-80%) of the studies were concerned with formulating a theory or implementing a concept. Furthermore, the most commonly used research method was conceptual analysis [Ramesh 2004] (i.e where the creator of some new technology makes a demo example in a suitable "native" context for that technology). In comparison, lab experiments represented less than 2% of the total, while case studies, data meta-analyses and field studies (covering a part of the software-intensive industry) each only represented 0.16%. Also, field experiments and surveys do not appear to be represented at all in the study by Ramesh et al. [Ramesh 2004]. Effort should therefore be made towards establishing a baseline for evaluation of new processes, technologies, platforms etc., and with standardized textual formats and guidelines for describing all relevant empirical artifacts, including subjects and objects. This could be achieved e.g. through accumulating experiences and lessons learned.

In our research we have used case study and survey as the principal research methods. The strengths and weaknesses of these two methods are therefore explored below.

Further discussion of Case study:

Utilizing a case study enables the researcher to prevent problems of scale and scope seen in small experiments [Kitchenham 1995]. It can thus be advantageous when the researcher has little or no control over the variables being investigated [Yin 2003].

Analytical generalization can be performed on the basis of case studies, i.e. generalization to a wider theory or application of theory, based on results from a set of several case studies [Yin 2003]. Furthermore, a case study implies no larger bias than any other research method [Flyvbjerg 2006].

The characteristics of case studies can furthermore be summarized [Flyvbjerg 2006] as:

- General theoretical knowledge is commonly seen as more valuable than context-bound knowledge.
- A single case study can be argued to contain one data point, hence it does not contribute to scientific advancement. However, sometimes even such studies have a large impact; consider the "Stanford Prison Experiment" [Haney 1973],

which had to be aborted after just a few days. Despite N (i.e. the number of such completed experiments) being less than 1 at the time, validity was still judged as satisfactory, since the outcome was in line with a 10,000 year-old history of master-slave relations.

- Nevertheless, developing or confirming general theories based on single case studies (i.e. external validation) is usually considered not possible, not even through multi-cases in the same company.
- Case studies are seen as more suitable for hypotheses generation, rather than testing hypotheses and building theory.
- Case studies are considered more easily biased towards validation of the results in support of the assumptions previously made by the researcher, e.g. during the design of the case study.

Also, industrial case studies are not so common due to the following [Kitchenham 1995] issues:

- Lack of access to critical information (insight or permission vs. confidentiality or interference conflicts).
- Duration of the study (vs. industrial commitment required).
- Instability impact (changes to the project during the study in duration, scope, personnel, environment).
- Context impact (obtaining permissions, effective communication).

Further discussion of Survey:

In a survey, generalization is commonly limited to the sample population [Robson 2002] [Fowler 2001] [Conradi 2005], thus sampling correctly from the target population is paramount (i.e. selecting a representative subset of the population). Important aspects of sample selection include the relevant population and parameters, as well as the sampling frame, method, size and cost [Cooper 2008]. The type of sampling can either be probability (i.e. each element in the population is given a non-zero chance of being selected) or non-probability (i.e. non-random and subjective selection of sample elements).

Surveys hold the promise of obtaining a large number of data points from a potentially well-defined population. They also allow combining qualitative and quantitative data collection. Furthermore, surveys also have a relatively low intervention cost while the study is on-going.

However, the cost in terms of time and effort to carry out the data collection process in a survey may prove very high compared to the response rate obtained [Conradi 2005]. This is partly due to the many levels of communication one may have to go through before actually reaching a potential respondent, in addition to several reminders and the fact that surveys are not being prioritized by the IT-industry. Furthermore, avoiding population-specific variations in the total process is also difficult, and may lead to uncontrollable biases in method.

Large-scale surveys also commonly yield a low response rate. As an example, the SEI carried out a survey on the state of practice for product family development, but only ended up with a response rate of merely 20% [Cohen 2002]. Furthermore, survey results reflect the opinions of the respondents regarding the phenomenon being investigated. These opinions may be biased, and also different from the actual population distribution [Kitchenham 2002].

A summary of strengths and weaknesses for the two research methods Case Study and Survey can be found in Table 2.

Table 2: Summary of Strengths and Weaknesses: Case Study and Survey

Research method	Strengths	Weaknesses
Case study	• Sum of cases encountered can be used towards context-independent knowledge (multi-case) • Multiple cases increases validity • Atypical or extreme cases are useful towards testing theories	• Context-bound knowledge commonly seen as less valuable • More suited towards generating than testing/building theory • Can be biased towards validation of results • Generalization to a wider theory / application is often difficult
Survey	• Large number of potential data points • Allows combination of quantitative and qualitative methods • Relative low cost of intervention	• Data collection time/effort may be high • Low response rate • Possible subjective bias

We chose case study as one of our research methods as this enabled us to directly contact specific companies in order to study industrial systems. At the same time, we were able to show that a study could be conducted in a structured manner and with specific benefits to the individual company. The knowledge gained was context-bound (which is generally a weakness of case studies), but is seen as a benefit here, yielding specific insights on the systems we were allowed to study. We were also able to tailor the research design towards industrial systems in a manner that was conducive to joint collaboration, thereby gaining easier access to the relevant data.

When investigating the evolutionary impact on development processes for in-house reusable and COTS/OSS components, as well as for our investigations on architectural risks and strategies, we chose to use survey as the research method. This was necessary to provide as large base for the data collection as possible; since information on nuances and larger differences in development processes is best obtained directly from the developers who use these processes. Surveys allowed us to use a relatively low amount of effort on collecting a high number of data points, while obtaining both qualitative and quantitative data.

2.8 Summary and the Challenges of this Thesis

Chapter 2 of this thesis has so far presented state-of-the-art and research challenges in Software Engineering, related to Software Quality and Process Improvement, CBSE

and Software Reuse, CBSE based on COTS and OSS components, Software Architecture, Software Maintenance and Evolution, and Software Risk Management. We have also discussed Research Methods in Software Engineering.

In this section, we further focus on those challenges that are directly relevant for the work in this thesis. These challenges are presented below as follows (called Research Challenges – RCs):

RC1: The impact of CBSE evolution on the development process: Challenges related to the development processes e.g. in terms of cross-dependencies, risks and ad-hoc tradeoffs are discussed earlier in this chapter. Although results from article P1 have also been reported in [Gupta 2009b] towards studying possible improvements to the actual reuse practice at StatoilHydro ASA, it is nevertheless important to investigate modernized processes and process changes for COTS/OSS and in-house reusable components. It is, for example, common to treat development involving reusable (in-house and external COTS/OSS) and non-reusable components in the same way. We would nevertheless expect distinct impacts on the development process when exploiting (in-house and external COTS/OSS) reusable components. This challenge is investigated by research question **RQ1:** *What is the state of practices and issues with respect to software process improvement in CBSE for COTS/OSS and in-house reusable software?*

RC2: The impact of CBSE evolution on software components: Defect density (as an indicator of reliability) and change density (as an indicator of maintainability) are important towards investigating the impact of CBSE-driven software evolution. Prior research has indicated that reusable components incur fewer defects than non-reusable components [Mohagheghi & Conradi 2004b]. Prior research on non-corrective changes has focused on number and type (i.e. preventive, perfective, adaptive) [Mohagheghi & Conradi 2004a]. While the results of these metrics on overall releases of reusable and non-reusable components were investigated by our research group and reported in [SP8] [SP9] [Gupta 2009b], investigating these metrics on the individual reusable components' level is also important. This allows for comparative analysis and provides a basis for SPI towards handling defects and changes in evolving CBSE systems. This challenge is addressed by research question **RQ2:** *How does software evolution impact individual reusable components, in terms of defect and change densities?*

RC3: Impact of Test Driven Development as an improvement to the software process: Earlier studies on the usage of TDD in industrial settings have shown, as a result, fewer defects (i.e. corrective changes) for non-reusable components [Janzen 2005]. Lower productivity has also been shown. However, neither TDD's effects on reusable components nor non-corrective changes with respect to TDD appear to have been explicitly considered in earlier empirical studies. Earlier research has indicated that predictability, stability and maintainability are more paramount for reusable components than for non-reusable components. It is therefore important to determine the effectiveness of Test Driven Development as an SPI to manage the risks associated with CBSE development of such components. This challenge is addressed by research question **RQ3:** *What are the impacts of Test Driven Development versus test-last development on reusable components?*

RC4: Architectural Software Evolution risks: Risks related to software evolution have previously been investigated by focussing on diagnosis results from structured evaluations, without taking software evolution or industrial adaptations specifically into account [Babar 2007a]. We expect that architectural risks caused by software evolution

in CBSE-driven systems will require specific management techniques. These techniques must be elicited and improved through alternative means (other than structured architecture evaluation outputs). We investigate this challenge in this thesis through research question ***RQ4:*** *What are the perceived architectural risks of CBSE-driven software evolution, and how can these risks be mitigated?*

3 Research Questions, Design and Implementation

We discuss here the research questions and their motivation based on the state-of-the-art. We also outline the context of the research questions, and further discuss how these were applied in practice. The research design and implementation, detailing the progression of the research, is also included.

3.1 Introduction

As mentioned in Chapter 1, we were involved with StatoilHydro ASA towards industrial case studies (**P3** and **P4**). StatoilHydro ASA is a large Oil & Gas company with huge amounts of data available and in active pursuit of university collaboration on relevant research topics, as evidenced in part by one of their key IT managers also holding an adjunct associate professor position at NTNU. Our collaboration with StatoilHydro ASA was thus achieved through mutual research interests, but required considerable attention to context detail to obtain the necessary data.

We used a survey in our collaboration with StatoilHydro to obtain additional qualitative information from the developers (**P1**), in order to complement and provide a qualitative base for the case study on reusable components in the company. We also used industrial surveys in our investigation on COTS/OSS in the European IT-industry (**P2**).

In investigating perceived architectural risks and corresponding management strategies in the Norwegian IT-industry, we first performed a pilot survey to elicit a set of relevant risks and strategies to be used as the starting point for a larger, more focused survey on these issues. Here too, survey as a research method was chosen since the information we sought could only be obtained directly from respondents, rather than from accumulated technical data. The surveys in articles **P2** and **P6** investigated larger samples than the other two (**P1** and **P5**). Nevertheless, we found that our practices and designs scaled up quite well, possibly influenced by prior survey experience.

3.2 Research Questions and their Motivation

In CBSE, software evolution/maintenance is an important research area of concern, because it encompasses changes that account for a large part of all software costs. These changes are necessary to modify CBSE components in a fast and reliable manner. They

cannot be avoided, but must be properly managed. Indeed, these changes provide the basis that allows software companies to take advantage of new opportunities and thereby stay competitive [Sommerville 2010]. The research questions defined for this thesis are described in the following sections.

3.2.1 RQ1: What is the state of practices and issues with respect to software process improvement in CBSE for COTS/OSS and in-house reusable software?

The first issue we investigate is the impact of "modern" CBSE on software development processes in CBSE, whether in-house or external (COTS/OSS). These impacts imply substantial alterations to existing software processes, shifts in focus during individual development process phases, or new techniques and tools being introduced. Additional empirical studies in industry are needed to validate the potential benefits of these new process changes, techniques and tools, especially as they also mean introducing new risks such as vendor control and integration issues. The possible advantages of using reusable components in software development are related to this. Furthermore, development of reusable components leads to a more complex development process, and requires additional organizational support [Crnkovic 2000].

Although reusable components also require alterations to the development process, they generally provide substantial benefits e.g. in terms of shorter time-to-market and lowered costs. Particular factors include whether reuse increases rework (problems caused by e.g. misunderstood or ambiguous requirements), and whether the reusable component information and quality specification is sufficient and trustable. Investigating these benefits and factors through industrial empirical investigations will contribute towards more targeted resource allocation, and improved handling of software evolution for reusable CBSE components. Our results from article P1 have been reported in [Gupta 2009b] as a basis for studying possible improvements to the actual reuse practice within StatoilHydro ASA. In this thesis, we use the results from article P1 to investigate the impact of modernized processes and process changes for in-house reusable components.

Here, we focus on obtaining qualitative data from developers by using survey interviews to explore these issues. Research question **RQ1** is explored through articles **P1** and **P2** in this thesis.

3.2.2 RQ2: How does software evolution impact individual reusable components, in terms of defect and change densities?

Here, we investigate the evolution impact in terms of occurrences and appearance of changes/defects in individual reusable software components. The number of change requests and defect reports returned from end-users and customers back to developers per time unit marks the most intensive periods of further development. This knowledge will also help to predict when peak resources are needed for future projects. These issues are related to **RQ1** in that they encompass resulting impacts at the level of individual components.

As aforementioned, earlier research has shown that reusable components incur fewer defects than non-reusable components [Mohagheghi 2004b]. Changes and defects on

the release level for both reusable and non-reusable software have been investigated by us in other investigations, which are reported in [Gupta 2009b]. Additionally, improving our knowledge and understanding of how software evolution impacts individual reusable components is important towards enabling targeted handling of these impacts and related issues.

The study of **RQ2** thus identifies how defects and changes evolve in individual reusable components. Relevant metrics are defect and change density, defined as number of defects or changes divided by NSLOC, respectively. **RQ2** is investigated through article **P3** in this thesis.

3.2.3 RQ3: What are the impacts of Test Driven Development versus test-last development on reusable components?

We further investigate traditional versus Test Driven Development with respect to the evolution impact on reusable components. Few empirical studies have investigated industrial systems developed using TDD methodology. These earlier investigations have often focused on defect reduction in relation to general software development [Janzen 2005], showing a decreasing defect density of 40-50% over non-TDD development, but for non-reusable software. They also show a change in development productivity, ranging from none or minimal to a 16% decrease, because of the added focus on writing tests prior to implementation. We are studying TDD in a software architecture restructuring/refactoring context, to discover whether improvements can be shown through empirical data on corrective and non-corrective changes.

Another earlier investigation on changes in reusable components in traditional test-last development [vanDeursen 2001a] showed that these components already exhibit a lower code modification density than non-reusable components (possibly due to their inherent higher maturity). Because of this, they may be less affected by changes than non-reusable components.

RQ3 focuses on determining the effectiveness of TDD in terms of defect and change densities (as defined in the previous section 3.2.2) to see impacts for reusable components. It is thereby related to **RQ2**, also through TDD's potentially beneficial impacts. **RQ3** is also connected to **RQ1** through the focus on TDD as an SPI practice. **RQ3** is explored through article **P4** in this thesis.

3.2.4 RQ4: What are the perceived architectural risks of CBSE-driven software evolution, and how can these risks be mitigated?

Our overall aim here is to investigate the possible perceived risks that affect the very architecture of a system, and how to manage and mitigate these risks. We then aim to suggest possible improvements by enabling a systematic approach towards architectural risk management in software evolution. This question is related to **RQ2**, in that it encompasses changes that require alterations to existing individual components and the architecture. It is also related to **RQ1** and **RQ3**, in that it focuses on the details surrounding the use of SPI towards the handling of CBSE risks in the IT-industry. However, **RQ4** more explicitly focuses on issues that can be perceived as risks to the continued success of the architecture. Furthermore, risks have potential impacts.

It is important to have strategies in place to handle possible risks up-front, so that these strategies are ready to be employed when the risks occur. The architectural properties of a system are commonly used towards predicting its resulting quality attributes (such as availability, performance, usability etc.) [Bass 2004]. These quality attributes are essential to the success of a software project, and can potentially be affected when the software architecture evolves. Also, the specific system attributes that provide evidence towards assessing these quality attributes are commonly left implicit [Bouwers 2009].

To explore research question **RQ4**, we have utilized surveys (published in articles **P5** and **P6**), as explained in the following Chapter 3.3. These aim to investigate the knowledge and experience of software architects through specific questions regarding perceived architectural risks.

3.3 Research Process Design and Implementation

The investigations in this thesis have been divided into three phases, grouping their contributions, as initially explained in Chapter 1 and mentioned later on in Chapter 5. Contributing to each other, these phases have been carried out with some overlap, due to the availability of data. We have chosen to use a mix of case studies and surveys. While case studies allow us to investigate detailed data on the company level, the practice of one company is limited and not suitable for generalization. Surveys were thus performed to investigate data on the industrial level. All of these have been conducted in the IT-industry. Data collection was performed on-site for the studies at StatoilHydro, while data collection on the surveys of the Norwegian and European industry was performed via web-enabled questionnaires and follow-up interviews.

The results and experience gained have been shared with the parties involved, to show clear benefits of the collaborations and to encourage future cooperative work. The investigations arranged by study type are as follows:

- **Phase 1 (RQ 1)** – industrial surveys (StatoilHydro and European IT-industry):
  - P1: A survey on developers' views on in-house software reuse (StatoilHydro)
  - P2: A survey of modern trends in development practices with COTS/OSS components (European IT-industry)

- **Phase 2 (RQ 2)** – industrial case study (at StatoilHydro):
  - P3: A case study on defect density and change density in individual reusable components

- **Phase 3 (RQ 3)** – industrial case study (at StatoilHydro):
  - P4: A case study of Test Driven Development in software evolution

- **Phase 3 (RQ 4)** – industrial surveys (both in the Norwegian IT-industry):

- P5: An exploratory survey on perceived risks and risk mitigation strategies regarding software architecture evolution
- P6: A full-scale survey on perceived risks and risk mitigation strategies regarding software architecture evolution, as well as proposing a tool (operational matrix) for software architecture risk management.

3.3.1 Industrial surveys in Phase 1: Developers' Attitudes (P1) and Development Practices (P2)

In phase 1, we first report on a survey at StatoilHydro, regarding in-house software reuse in the company. The company is a large, Norwegian company in the Oil & Gas industry. It is represented in 25 countries, has a total of 28,000 employees, and is headquartered in Europe. The company's central IT-department is responsible for developing, delivering and operating software to flexibly aid key business areas. There are approximately 100 developers and consultants in this department, most of them located in Norway.

Information in terms of e.g. benefits experienced, problems encountered, and possible improvements towards future reuse are not easily obtained from studying technical data alone. The first survey in Phase 1 was thus performed specifically to elicit developers' views on software reuse. The survey counted only 16 respondents (all at StatoilHydro), but nevertheless all the relevant developers and roles were involved. Our survey at StatoilHydro led to the publication of **P1**.

We have carried out a second much larger survey study externally in the IT-industry. It was performed to investigate development practices regarding COTS and OSS components. The aim was to investigate the IT-industry state-of-practice by obtaining information directly from developers with relevant experience and knowledge. This survey counted 133 respondents from 127 companies, spanning the IT-industry in Norway, Germany and Italy. It was followed up by semi-structured interviews with 28 of the respondents. A full review of the context in this survey is published in [SP5]. Article **P2** summarizes our results and experiences from this second study, and a more detailed discussion of the individual results can be found in the secondary articles.

Articles **P1** and **P2** were selected towards answering **RQ1:** *What is the state of practices and issues with respect to software process improvement in CBSE for COTS/OSS and in-house reusable software?*

3.3.2 Industrial case studies: Phase 2 – Defect and Change Densities (P3) and Phase 3 – Test Driven Development (P4)

The IT strategy within StatoilHydro on reuse was initiated in response to changing business and market trends, to provide a consistent and resilient technical platform for software development and integration [O&S 2006]. It is now being expanded towards other divisions within Statoil ASA.

For our first investigation, in phase 2, the actual JEF framework consisted of seven separate components (ranging in size from 181 to 8885 NSLOC of Java code, with a total of three releases), which can be applied separately or collectively towards application development.

Later, in phase 3, we studied two additional releases of the JEF framework. These two releases were developed using Test Driven Development, and comprise only five components due to a refactoring; it was determined that a shift of focus was needed to improve reusability of the architecture, in terms of components used and services provided.

All work on JEF prior to release 1 was for department-internal development and testing only, while release 1 was the first to be used in other development projects in Statoil ASA. In our investigations of the framework, we have used applications reusing it in new development for comparison.

Two of the most important data sources were change requests (CRs) and trouble reports (TRs) in StatoilHydro ASA, as they are part of the stated quality focus for their reuse program. We now briefly discuss how these two are handled within the company.

Change requests in StatoilHydro ASA:

When a change (in a requirement) is identified, a CR is established and registered in the Rational ClearQuest tool. Examples of change requests are:

- adding new or modifying existing functionalities, or enhancing performance (perfective changes)
- improving maintainability for the future (preventive changes)
- adapting to changes in environment or platform, e.g. related to other JEF component interfaces (adaptive changes)
- (corrective changes are dealt with through trouble reports, described below)

A change request normally impacts only one of the JEF components, but may impact several. If a change request impacts several components, it is related to the category *General* to indicate that this change request impacts the JEF framework as a whole (or that it cannot be assigned to one specific component alone). All registered change requests can be exported as Microsoft Excel files.

Each change request contains an ID, headline description, priority (of the change request) given by both the customer and the developer (Critical, High, Medium or Low), estimated duration to solve, remaining duration to solve, subsystem location (one of the seven JEF components), system location (i.e. JEF framework, non-reusable application, etc.), as well as an updated action and timestamp record for each new state the change request goes through.

StatoilHydro ASA does not register release numbers for changes, but a change request is always marked with a timestamp at registration. This timestamp is consistent with the release that was currently under development at the time. Also, effort is not explicitly registered in the system, not even as "small, medium or large". Only 'remaining duration' vs. team size is sporadically used for rough estimates of effort required.

Trouble reports in StatoilHydro ASA:

The process for handling defects is very similar to that for changes. When an assumed defect (i.e. an assumed execution failure) is found during integration/system

testing or execution (in post-release operation), a TR is established and registered in the Rational ClearQuest tool. A defect also usually impacts only one of the JEF components, but as with CRs, it may impact more than one. When this is the case, it is similarly named *General*. All registered trouble reports can likewise be exported as Microsoft Excel files.

Each trouble report record contains an ID, headline description, priority (regarding the technical aspects and given by the developer as Critical, High, Medium or Low), severity of the problem (given by the customer or end user as Critical, High, Medium or Low), state classification (Error, Error in other system, Duplicate, Rejected or Postponed), estimated duration to fix, remaining duration to fix, subsystem location (one of the JEF components), system location (e.g. JEF, DCF), as well as an updated action and timestamp record for each new workflow state. As with CRs, TRs do not include release numbers, but contain timestamps for time of registration. They also do not include effort information, but include an estimate for the duration to fix a given defect.

An advantage of the close similarities between trouble reports and change requests is that Statoil personnel can easily switch from working with one to the other without extra training. This saves effort and resources when the reshuffling of responsibilities becomes necessary.

Metrics in StatoilHydro ASA:

To initiate the collaboration with StatoilHydro ASA, we started with approaching the questions of "what" and "how" to measure. Through this approach, we were able to combine the quality foci of the company, in terms of defect and change density, with our research goals towards software evolution within CBSE (**RQ2**). We chose to use a case study approach to investigate these data, since they were of a longitudinal nature and therefore well suited towards studying software evolution in an industrial (case) setting.

Our investigations on software evolution started with a study on reusable components. The results from this investigation are presented in article **P3**. We then further studied software changes made to these components in more detail. These reusable components were (and are) still being refined and further developed. This made it possible for us to establish another more longitudinal case study on the possible advantages and disadvantages of TDD versus traditional "test-last" development methodology on these reusable components. The results of this case study are presented in **P4.**

P3 was selected towards answering **RQ2:** *How does software evolution impact individual reusable components, in terms of defect and change densities?*

P4 was selected towards answering **RQ3:** *What are the impacts of Test Driven Development versus test-last development on reusable components?* Further results from our case studies with StatoilHydro ASA can be found in the secondary articles **SP8** and **SP9**.

In investigating **RQ2** and **RQ3**, we used the following metrics:

- *defect density*, defined as the number of defects (TRs) divided by NSLOC, and
- *change density*, defined as the number of changes (CRs) divided by NSLOC.

3.3.3 Industrial surveys in Phase 3: Perceived Software Architecture Evolution Risks (P5, P6)

Our aim was to investigate perceived risks and risk management strategies specifically for software architecture evolution. Earlier studies in this area have focused on quantitative outputs from software architecture evaluation methods accumulated over time. However, use of such evaluation methods is uncommon in the industry. Rather, practiced evaluation methods range from fully structured to completely ad-hoc and informal [Babar 2007a]. To obtain the full spectrum of information, and gain risks and strategies possibly missed by earlier studies, we chose to use a survey approach here also, so that we could involve actual software architects.

We performed a pilot survey to obtain initial data collection and calibration. Here, we used a convenience sample of 16 IT-professionals in different Norwegian companies with prior knowledge and experience with software architecture, employing semi-structured interviews for data collection (published in **P5)**.

Based on the outcome and experience gained from this pilot survey, the full-scale survey was then run in the software-intensive IT-industry in Norway, and published in **P6**. In this survey, the sample size was expanded towards a larger portion of the software-intensive IT-industry in Norway (resulting in a total of 82 completed responses from 511 potential respondents). In anticipation of the larger amounts of data, the survey data collection was also altered to use a questionnaire by web-interface.

The systems studied in these two investigations in phase 3 deal with software systems that have two major characteristics:

- use of CBSE, and
- changes in the systems' software architectures during their lifetimes.

This entails development projects that have at least delivered the first production release, i.e. can be said to be in the ‘maintenance / evolution’ phase.

Articles **P5** and **P6** contribute towards answering **RQ4:** *What are the perceived architectural risks of CBSE-driven software evolution, and how can these risks be mitigated?*

Research question **RQ4** was therefore (like **RQ1**) explored through two successive surveys, in order to obtain answers based on the skills, knowledge and experience of software architects, focusing on perceived architectural risks related to software evolution (where all respondents have done related implementation work).

4 Results

We here outline our results from the viewpoint of research phases, as outlined in the earlier Figure 1 (Chapter 1), presenting each study individually. Thereafter, we summarize the contributions of this thesis in connection with the individual studies.

4.1 Results from the individual research phases

Our phases are here summarized by investigation results pertaining to the respective research questions, connection to contributions, and overall theme for this thesis. The investigations entailed in each phase are further presented in terms of contributions.

4.1.1 Phase 1 contributions to C1 from article P1: An Empirical Study of Developers Views on Software Reuse in Statoil ASA

C1: Improved knowledge of modern trends in CBSE and their impacts on software development processes

Phase 1 entailed studying old non-reuse vs. new reuse-centric development processes. In article **P1**, we studied a reuse program in the IT-department of a large industrial company (StatoilHydro ASA), with their reusable Java framework (JEF) and surrounding processes.

We wanted to obtain qualitative information related to our results from investigating the company's reuse program. To accomplish this, we surveyed issues related to software reuse, from the viewpoint of developers who are involved in the reuse program. We focused on exploring the possible benefits, disadvantages, and contributing factors that characterize successful reuse of software components. The investigation also encompasses the documentation and quality specifications for reusable components, and this material is available to developers when they employ reusable components in new development. Our results from this investigation convey the opinions of developers regarding software reuse, in the following five main areas:

- The **benefits of reuse** are viewed as **lower costs, shorter development time, higher quality** of the reusable JEF components, and **a more standardized architecture**. This is in keeping with results from literature [Lim 1994].
- On **factors contributing towards reuse**, **no link to level of education or experience** was seen. The findings on formal processes appear to support prior research [Frakes 1995]; a **formal process for general software development** may have an implicit positive effect. Also, the results show that **improvement**

of the documentation of the reusable components would be very advantageous towards achieving successful reuse.

- **No** statistically significant **relations between reuse and increased rework** were found, and so no conclusion can be reached on this issue. A **possible cause** is the **mandate of reuse** in the company (i.e. reuse of JEF components is required in all new development, as decided by upper level management), as well as the existence of multiple responsibility roles that commonly cross project- and team-lines. Due to these issues, there is **no clear-cut division between *development for/with reuse*** in the company.
- Most developers have adequate **understanding of the components.** However, it is mainly the **JEF team and prior experience** (not training) that is being **used by the developers** to achieve this level of understanding.
- **Quality attribute specifications** for the development projects ***developing with reuse* are trusted**, whereas **for the reusable components they are insufficient**.

4.1.2 Phase 1 contributions to C1 from article P2: Development with Off-The-Shelf Components: 10 Facts

C1: Improved knowledge of modern trends in CBSE and their impacts on software development processes

In article **P2**, we investigated the state-of-the-practice in the European IT-industry concerning development based on COTS and OSS components, using a large multi-national survey in Norway, Germany and Italy**.**

Here, we also investigated COTS/OSS in the IT-industry. The results from this study are presented as a set of findings regarding industrial practices with development based on Off-The-Shelf (OTS, i.e. COTS and OSS) components. The findings are taken verbatim from **P2** and numbered as Fact 1 – 10, with corresponding complementary summaries:

"Fact 1 – Development process: *Companies use traditional processes enriched with OTS-specific activities to integrate OTS components."* Familiarity with OTS candidate components is an important factor to consider in customizing the entire development process. Sufficient knowledge of OTS candidate components may make the use of already adapted development processes (e.g. adapted evolutionary) unnecessary.

"Fact 2 – Component selection: *Integrators select OTS components informally. They rarely use formal selection procedures."* Benefits of and pre-conditions for using a formal component selection process are unclear due to the lack of clear empirical evidence. Lacking such evidence leaves integrators reluctant towards using such a formal process since it is also presumed complex and time-consuming.

"Fact 3 – Component selection: *There is no specific phase of the development process in which integrators select OTS components. Selecting components in early phases has both benefits and challenges."* We have identified possible problems that component integrators must consider when selecting OTS components in the early phases of a software development project.

"Fact 4 – Component integration: *Estimators use personal experience when they estimate the effort required to integrate components and most of the time they do not estimate accurately. Stakeholder-related factors will affect dramatically the accuracy of estimates."* Some estimation tools, e.g. COCOTS [Abts 2000], consider both the technical nature of the components, and e.g. component understandability and vendor response time. Estimation tools should also take into account possible requirement changes and the evolution of components, especially for large, long-lived projects.

"Fact 5 – Quality of the integrated system: *Negative effects of OTS components on the quality of the overall system are rare."* For reasons such as low costs, component integrators sometimes select OTS components of a lower quality. It is thus the quality assurance efforts during selection and integration that ensure the quality of the OTS component in the finished system.

"Fact 6 – OSS and COTS components: *Integrators usually used OSS components in the same way as commercial components, i.e. without modification."* Alterations to the source code of an OSS component may be infeasible, in particular for long-lived commercial systems with many evolutionary iterations over their lifetimes, due to the possible internal support required. The context of the application must therefore be considered when deciding whether to use OTS components.

"Fact 7 – Locating defects is difficult: *Although problems with OTS components are rare, the cost of locating (i.e. within or outside OTS components) and debugging defects in OTS-based systems is substantial."* The deployment environments and configurations of OTS components come in a wide variety. This variety represents an obstacle towards reproducing reported errors, and irreproducible errors will commonly not be prioritized to be fixed by the component provider.

"Fact 8 – Relationship with the provider: *The relationship with the OTS component provider involves much more than defect fixing during the maintenance phase."* Different persons within an OSS community may be involved in separate tasks supporting the use (e.g. evaluation, selection, integration) of a component.

"Fact 9 – Relationship with the client: *Involving clients in OTS component decisions is rare and sometimes unfeasible."* It is often the case that the application client has no interest in implementation technicalities, due to lack of relevant competencies. It is thus important that clients' interests and technical competences are clarified at the start of a project to determine possible strategies for requirements (re)negotiation.

"Fact 10 – Knowledge management: *Knowledge that goes beyond the functional features of OTS components must be managed."* External channels for sharing knowledge and experience on the use of COTS/OSS are few and uncommon. This kind of information is spread across e.g. portals and bulletin boards, or managed internally by a 'component responsible'. It is therefore essential to manage knowledge beyond mere component functionality.

Furthermore, we have found that there is a mismatch between academic theory (which is often based on incorrect assumptions) and industrial practice when it comes to components usage, due to the lack of empirical evidence, the lack of studies involving industry, and the lack of industrial adoption of research results. Examples include:

- While academia has deemed traditional development models unsuitable for COTS/OSS development and calls for adapted models, companies simply enrich these with COTS/OSS-relevant activities since they have sufficient knowledge of relevant COTS/OSS components.
- Researchers suggest selecting COTS/OSS components early in the development cycle, but there is no specific phase for this activity in industry. Moreover, selecting components early has both benefits and challenges.

4.1.3 Phase 2 contributions to C2 from article P3: Preliminary results from an investigation of software evolution in industry

C2: Improved understanding of evolution impact on individual reusable components in terms of defect and change densities

Our case study here was on the evolutionary behavior of the quality attributes defect density and change density for individual reusable components (defined as units of composition, specified such that their interfaces are separate from their implementation [Crnkovic 2002]). Prior research had shown that reusable components were more stable (i.e. that they have a lower rate per NSLOC of code modification) over several releases [Mohagheghi & Conradi 2004b].

Our results on individual reusable components showed that the components investigated had lower defect densities over several releases. Moreover, although the larger components had higher defect densities in the first release, this trend did not continue over several releases. Five of the six reusable components had a higher change density in the first release than in subsequent releases. However, in subsequent releases the larger components no longer had the higher change densities.

In summary, we verified findings on defect and change density for individual reusable components that were part of software development at a large industrial company in Norway (StatoilHydro), while they experienced new and changed requirements through software evolution. These components have a decreasing defect density over several releases, but for change density the results remain inconclusive.

4.1.4 Phase 3 contributions to C3 from article P4: The Impact of Test Driven Development on the Evolution of a Reusable Framework of Components – An Industrial Case Study

C3. Improved understanding of the impact and effectiveness of TDD

The investigation on development method compares defect and change densities for traditional test-last development with that of TDD, and also investigates the relation between the two metrics as well as the distribution of changes for the TDD approach. The new approach was introduced for the development of the latest two releases of the reusable framework of components in the company, in order to facilitate improvements in the architecture of the reusable components framework. Our results show that the mean defect density and the mean change density per release were both reduced when using the TDD approach; the former by 35.86% and the latter by 76.19%. However,

these effects appear to change drastically over several releases; the defect density exhibiting the most substantial changes. We also did not see any indication that the architecture was negatively impacted, though this is mentioned as a possible disadvantage in other research [Foote 1997] [Perry 1992] [George 2004]. Finally, the distribution of changes was heavily skewed towards preventive changes, showing the effects of the refactoring inherent to TDD.

In summary, the evolution of reusable components was further explored in relation to the impact of development approach. TDD was shown to lead to lower mean defect density and mean change density for reusable components over traditional test-last development.

4.1.5 Phase 3 contributions to C4a from article P5: Identifying and Understanding Architectural Risks in Software Evolution: An Empirical Study

C4a: Identification of perceived risks and related mitigation strategies specifically for the evolution of software architecture

To investigate issues directly related to risk management in software architecture evolution, we first elicited actually experienced risks and employed mitigation strategies from software architects in industry. This was performed through a pilot survey entailing a series of semi-structured interviews, and the results are presented in article **P5**.

The risks and management strategies we discovered in our pilot study were summarized and used as input towards the questionnaire-based full-scale survey. We investigated risks and strategies on the technical, process and organizational levels. Technical issues (e.g. in terms of existing or new technologies) can affect the architecture of a system to a large extent. Also, organizational and process issues are important because they are central to the success of operative reuse programs in industry.

Our results from this first software architecture study show that the most influential risk was that *"lack of stakeholder communication influenced the implementation of new and changed architectural requirements in a negative way"*. This was also the most frequent risk encountered by the respondents. Secondly, *"poor functionality clustering causing disadvantages towards performance"* was also seen among the most frequent risks. Additionally, we found that there is little effort among software architects to evaluate and document the architecture, as they attempt to meet challenges as they are encountered during development.

4.1.6 Phase 3 contributions to C4a from article P6: An Empirical Study of Architects' Views on Risk Management Issues for Software Evolution

C4a: Identification of perceived risks and related mitigation strategies specifically for the evolution of software architecture

A larger investigation was warranted to further investigate risks and management strategies towards software architecture evolution, versus our initial study of perceived

architectural risks in article **P5.** Our results from this full-scale survey (article **P6**) showed that:

- The overall most influential risk dealt with poor functionality clustering causing performance problems.
- The corresponding most successful mitigation strategies were to refactor the architecture, and to design with a high focus on modifiability.
- The second most influential risk was that insufficient stakeholder communication affected requirements negotiation and implementation of new / changed architectural requirements in a negative way.
- The most successful strategies towards this risk were to increase team communication efforts and to arrange stakeholder plenary meetings.

Therefore, the risks that were identified as the two overall most influential in the pilot survey were also identified in the full-scale survey.

Most of the risks we identified were placed into the following categories, considered in related work: Requirements risks, Architecture Team risks, and Stakeholder risks (from the subcontractors' viewpoint) from Ropponen et al. [Ropponen 2000], as well as Quality Attribute risks, Integration risks, Requirements risks, Documentation risks, Process and Tools risks, Allocation risks, and Coordination risks from Bass et al. [Bass 2007]. Nevertheless, the following three risks we identified (listed below as <risk - consequence>) do not appear to fit these categories from literature:

- Extensive focus on streamlining of the architecture - affected modifiability negatively (technical risk TR 4)
- Lack of business context analysis - affected stakeholder relationships negatively (process risk PR 7)
- Prior architecture maintenance/evolution pushed to other projects due to lack of personnel - influenced knowledge on the architecture negatively (organizational risk OR 5)

4.1.7 Phase 3 Contributions to C4b from article P6: An Empirical Study of Architects' Views on Risk Management Issues for Software Evolution

C4b: An adapted operational matrix as a tool to support risk management in software architecture evolution

We further developed a three-part **operational matrix** of risks and corresponding mitigation strategies as a tool to support risk management in software architecture evolution, based on the identified risks and strategies under contribution **C4a**. This matrix is based on the levels of technical, process and organizational risks, and includes an aggregated rating of outcome towards successful risk mitigation for each strategy.

The matrix is presented here in three parts for **technical (Table 3), process (Table 4) and organizational risks (Table 5)**, over the following pages**.** This matrix shows identified risks based on indicated level of influence, and corresponding management strategies along with a relative ranking of outcome towards successful mitigation, and can be expanded as needed. Risk influence is indicated as the number of respondents who replied that the corresponding risk had a "Very High" (VH) or "High" (H) influence on the architecture.

Table 3: Adapted operational matrix for the most influential Technical Risks (VH > 1) and corresponding management strategies in software architecture evolution

Technical (identified in planning), ID: Risk	**Risk Influence**	**ID:Strategy:Outcome rating =** Number of {"Not at all", "Somewhat", "Medium", "Mostly", and "Completely"} successful instances.		
TR 1: Poor clustering of functionality affected performance negatively *	VH: 7, H: 23	**TS 1**	Refactoring of the architecture	{0, 8, 2, 5, 3}
		TS 2	Redesign within constraints	{0, 0, 1, 4, 0}
		TS 3	Design with high focus on modifiability	{0, 1, 2, 6, 1}
		TS 4	Finalize modifiability design considerations early	{0, 1, 0, 0, 0}
TR 2: Requirements from other system(s) affected performance negatively	VH: 5, H: 10	**TS 2**	Redesign within constraints	{0, 1, 4, 4, 0}
		TS 5	Employ separate agents for external communication, protocol for information sharing	{0, 1, 2, 2, 0}
		TS 3	Design with high focus on modifiability	{0, 1, 4, 3, 0}
TR 3: Undefined variation points in requirements affected performance negatively, caused increased focus on modifiability	VH: 3, H: 10	**TS 3**	Design with high focus on modifiability	{0, 0, 3, 5, 1}
		TS 4	Finalize modifiability design considerations early	{0, 3, 2, 3, 0}
TR 4: Extensive focus on streamlining of the architecture affected modifiability negatively	VH: 2, H: 10	**TS 3**	Design with high focus on modifiability	{0, 0, 3, 3, 1}
		TS 4	Finalize modifiability design considerations early	{0, 2, 3, 3, 0}
TR 5: Architectural mismatch caused redesign of part of the architecture	VH: 2, H: 2	**TS 1**	Refactoring of the architecture	{0, 1, 1, 0, 0}
		TS 3	Design with high focus on modifiability	{0, 0, 0, 1, 0}
		TS 4	Finalize modifiability design considerations early	{0, 0, 1, 0, 0}
Experienced during				
TR 6: Increased focus on modifiability contributed negatively towards system performance *	VH: 6, H: 10	**TS 6**	Implementation of changes towards improved modifiability	{0, 0, 2, 1, 0}
		TS 7	Minor implementation changes	{0, 1, 6, 7, 0}
TR 7: Poor original core design prolonged the duration of the maintenance/ evolution cycle *	VH: 3, H: 11	**TS 6**	Implementation of changes towards improved modifiability	{0, 0, 3, 4, 0}
		TS 8	Informal review of the architecture	{0, 0, 3, 3, 0}
		TS 7	Minor implementation changes	{0, 0, 1, 2, 0}
		TS 1	Refactoring the architecture	{0, 0, 3, 0, 0}
TR 8: Varying release cycles for COTS/OSS components made it difficult to implement required changes *	VH: 2, H: 16	**TS 9**	Use own development as potential backup solution	{0, 4, 5, 8, 0}
		TS 10	Implement extra architecture add-ons	{0, 1, 2, 0, 0}

Table 4: Adapted operational matrix for the most influential Process Risks (VH > 1) and corresponding management strategies in software architecture evolution

Process (identified in planning), ID: Risk	**Risk Influence**	**ID:Strategy:Outcome rating =** Number of {"Not at all", "Somewhat", "Medium", "Mostly", and "Completely"} successful instances.		
PR 1: Lack of architecture documentation required more effort to be spent on planning during maintenance/evolution *	VH: 6, H: 25	**PS 1**	Recover needed architecture documentation using current architecture design and other artefacts as a basis	{0, 3, 2, 5, 1}
		PS 2	Thorough planning before implementing maintenance/evolution changes	{0, 1, 8, 7, 1}
		PS 3	Recover architecture evaluation artefacts where needed	{0, 0, 4, 2, 0}
		PS 4	Alter process to capture important architecture details	{0, 1, 3, 3, 0}
		PS 5	Explicit training on architecture documentation	{0, 0, 1, 3, 0}
PR 2: Lack of architecture evaluation contributed to discovering potential problems later in planning of maintenance/evolution	VH: 5, H: 13	**PS 1**	Recover needed architecture documentation using current architecture design and other artefacts as a basis	{0, 0, 3, 4, 1}
		PS 3	Recover architecture evaluation artefacts where needed	{0, 1, 2, 4, 0}
		PS 4	Alter process to capture important architecture details	{0, 0, 2, 3, 0}
PR 3: Lack of business context analysis affected stakeholder relationships negatively	VH: 4, H: 13	**PS 6**	Integrate business context in planning of the maintenance/evolution	{0, 2, 5, 3, 1}
		PS 7	Include business context informally	{0, 1, 1, 4, 0}
PR 4: Insufficient requirements negotiation postponed important architecture decisions	VH: 4, H: 9	**PS 8**	Negotiate requirements early	{0, 0, 2, 2, 1}
		PS 9	More explicit communication	{0, 2, 3, 0, 0}
		PS 10	Allow additional time for communication and feedback	{0, 1, 1, 3, 0}
Experienced during				
PR 5: Insufficient stakeholder communication contributed to insufficient requirements negotiation and affected implementation of new/changed architectural requirements negatively	VH: 7, H: 13	**PS 13**	Extra communication effort	{0, 1, 7, 3, 0}
		PS 14	Postpone some requirements to next maintenance/evolution cycle	{0, 0, 1, 2, 0}
		PS 15	Arrange plenary meetings for all stakeholders	{0, 0, 3, 4, 0}
		PS 16	Negotiate project extension	{0, 1, 2, 2, 0}
PR 6: Poor integration of architecture changes into implementation process affected implementation process and the architecture design negatively *	VH: 2, H: 20	**PS 17**	Overlay architecture change process onto implementation of maintenance/evolution	{0, 0, 4, 7, 1}
		PS 18	Integrate architecture considerations into implementation process	{0, 1, 9, 2, 0}

Table 5: Adapted operational matrix for the most influential Organizational Risks (VH > 1) and corresponding management strategies in software architecture evolution

Organization (identified in planning), ID: Risk	**Risk Influence**	**ID:Strategy:Outcome rating =** Number of {"Not at all", "Somewhat", "Medium", "Mostly", and "Completely"} successful instances.		
OR 1: Not allowed to change OSS as decision mandate external to architecture team, affecting performance and modifiability negatively *	VH: 6, H: 22	**OS 1**	Frequent, interactive, scheduled meetings to keep up to date	{0, 1, 4, 5, 0}
		OS 2	Involve all "layers" of customer organization as stakeholders, allow extra time for proper communication	{0, 0, 1, 0, 0}
		OS 3	Ensure compliance with external mandate holder	{0, 0, 4, 1, 0}
		OS 4	Involve mandate holder early as stakeholder in planning	{0, 2, 4, 9, 1}
OR 2: Separate architecture team per maintenance/evolution cycle basis contributed to loss of and insufficient knowledge about the existing architectural design *	VH: 4, H: 29	**OS 5**	Dedicate personnel to "retrieve" architecture knowledge	{0, 2, 11, 6, 0}
		OS 6	Increased focus on proper documentation, to allow bringing new personnel up to speed quickly	{0, 1, 8, 6, 0}
OR 3: Cooperative maintenance/evolution with architects from customer organization required extra training and communication efforts *	VH: 3, H: 10	**OS 1**	Frequent, interactive, scheduled meetings to keep up to date	{0, 0, 1, 1, 0}
		OS 7	Perform maintenance/evolution incrementally	{0, 0, 2, 0, 0}
		OS 8	Allot extra time for proper communication with all stakeholders	{0, 0, 1, 0, 0}
		OS 9	Include other project's architects in planning, implementation	{0, 1, 4, 5 0}
OR 4: Lack of clear point of contact from customer organization contributed to inconsistencies in communication of the architecture and requirements *	VH: 2, H: 27	**OS 5**	Dedicate personnel to "retrieve" architecture knowledge	{0, 0, 1, 0, 0}
		OS 1	Frequent, interactive, scheduled meetings to keep up to date	{0, 0, 4, 5, 0}
		OS 2	Involve all "layers" of customer organization as stakeholders, allow extra time for proper communication	{0, 3, 6, 4, 1}
		OS 6	Increased focus on proper documentation, to allow bringing new personnel up to speed quickly	{0, 1, 5, 6, 0}
Experienced during				
OR 5: Prior architecture maintenance/evolution pushed to other projects due to lack of personnel influenced knowledge on the architecture negatively *	VH: 3, H: 11	**OS 10**	Regain architecture details from remaining upper management personnel	{0, 0, 2, 1, 0}
		OS 11	Keep architecture documentation centralized	{0, 0, 5, 8, 0}
OR 2: Separate architecture team per maintenance/evolution cycle contributed to loss of and insufficient knowledge about the existing architectural design *	VH: 2, H: 13	**OS 10**	Regain architecture details from remaining upper management personnel	{0, 2, 6, 6, 0}
		OS 11	Keep architecture documentation centralized	{0, 0, 0, 1, 0}
		OS 12	Set up standard procedure for distribution of architecture documentation and knowledge	{0, 2, 0, 0, 0}

4.2 Summary of research phases

Tables 6 and 7 below summarize the findings of this thesis from the preceding chapters, relating research questions, articles, contributions, research methods, validity observations, aftermath reflections and phases to each other.

Table 6: Relations between SEVO goals, research phases, research questions and articles

SEVO Goals	**G1:** Better understanding of software evolution, especially for CBSE. **(G3+G4)**		**G2:** Better methods to predict the risks, costs, and profile of software evolution in CBSE. **(G3+G4)**	
Research Phase	**Research Phase 1**	**Research Phase 2**	**Research Phase 3**	
Research Question (RQ)	**RQ1:** What is the state of practices and issues with respect to software process improvement in CBSE for COTS/OSS and in-house reusable software?	**RQ2:** How does software evolution impact individual reusable components, in terms of defect and change densities?	**RQ3:** What are the impacts of Test Driven Development versus test-last development on reusable components?	**RQ4:** What are the perceived architectural risks of CBSE-driven software evolution, and how can these risks be mitigated?
Articles	**P1, P2**	**P3**	**P4**	**P5, P6**

Table 7: Relations between articles, contributions, research methods, validity observations, and aftermath reflections

Articles	**P1, P2**	**P3**	**P4**	**P5, P6**
Contributions	To **C1:** Identification of major impacts of modern trends in CBSE on the development process for in-house reusable and COTS/OSS components.	To **C2:** Individual reusable components were shown to have a decreasing defect density over several releases.	To **C3:** Test Driven Development led to lower mean defect density and mean change density for reusable components over traditional test-last development.	To **C4a:** Identification of a set of actual risks experienced, and corresponding mitigation strategies used, by software architects.
	To **C1:** Discrepancies between academic theory and industrial practice were identified.	To **C2:** Decreasing change densities were shown for five of six components over several releases.		To **C4b:** Development of a three-part adapted operational matrix as a tool to support risk management in software architecture evolution.
Research Methods	Survey, followed up by semi-structured interviews	Case study	Case study	Survey
Validity Observations (Chapter 5 of this thesis)	The questionnaire questions are based on the research literature.	Both defect and change density are described and used in the research literature.	Both defect and change density are described and used in the research literature.	The questionnaire questions are based on the research literature.
Aftermath reflections	Surveys in industry require close follow-up, and large amounts of resources, while returning a relatively low response rate.	Studying industrial systems requires close connections with the developing organization.	Studying industrial systems requires close connections with the developing organization.	Surveys in industry require close follow-up, and large amounts of resources, while returning a relatively low response rate.

5 *Evaluation and Discussion*

This chapter discusses our four research questions (RQ1, RQ2, RQ3 and RQ4), based on the results presented in Chapter 4. We also discuss the relationships between the contributions, research questions and results, together with a more in-depth discussion of the observed results. The relationships between our contributions and the state-of-the-art, as well as the overall SEVO project research goals, are then considered, together with general contributions to StatoilHydro. We further summarize the validity of the individual studies behind our contributions. Finally, we also include a reflection on the context of our research project in this chapter.

5.1 RQ1: What is the state of practices and issues with respect to software process improvement in CBSE for COTS/OSS and in-house reusable software?

Contribution C1: Improved knowledge of modern trends in CBSE and their impacts on software development processes; articles P1, P2

Impact for in-house reusable components: We interviewed the developers involved in the reuse program at StatoilHydro ASA to identify and analyze the possible impacts on the development process.

- **A defined / standardized architecture is seen as key:** It is likely, however, that the benefits of such standardization may be short-lived, as the architecture will probably also need to be evolved in order to accommodate future changes. This also provides a starting point for investigating architectural risks in CBSE-driven software evolution.

- **Organization of knowledge sharing remains important:** Improved organization of knowledge sharing is needed; senior personnel in the company indicated that some external consultants were added to the project after regular training activities were carried out. These consultants therefore experienced a lack of knowledge with regards to the reuse practices of the company. This could be helped through improvement of documentation of the reusable components (as the results show that a managed collection of **information**

would be beneficial). Qualitative data from developers on related benefits includes improved information sharing and learning, higher documentation quality, better overview of functionality, and access to FAQ (frequently asked questions) answers. Improved organization of training and knowledge sharing would also aid in a better understanding of the quality specifications of the reusable components – indicated as a problem potentially caused by rapid changes in requirements, resources and personnel. Nevertheless, the developers appear to have a good understanding of the reusable components themselves, obtained through informal knowledge sharing.

Impact for external COTS/OSS components: When it comes to external COTS/OSS (both called OTS – Off-The-Shelf) components, impacts in several dimensions can be seen in that traditional processes, enriched with OTS-specific activities, are being used to select and integrate OTS components as follows:

- **Selection:** Selection of OTS components is done informally, without specific focus on a particular lifecycle phase. More systematic management of OTS components knowledge beyond functional features is, however, necessary.
- **Integration and testing:** It becomes costly to debug defects in the borderland between in-house and external components, where the latter are mainly treated as "black boxes" (closed source). Moreover, since a multitude of test criteria for both functional ("black box") and structural now exist, the focus is more on determining the right combination of criteria towards efficient testing [Bertolino 2007].
- **Effort Estimation:** Such estimation is performed using personal experience, commonly inaccurate, and largely affected by stakeholders.
- **Traditional Quality:** Negative quality effects are rare, and the trust in external components high.
- **Management complexity:** There is a complex relationship between an application developer and an OTS provider, while customers (new software owners) are commonly not involved in decisions about use of certain OTS components. (Re)negotiation of requirements is therefore nonexistent, as stated in **P2.**

5.2 RQ2: How does software evolution impact individual reusable components, in terms of defect and change densities?

Contribution C2: Improved understanding of evolution impact on individual reusable components in terms of defect and change densities; article P3

- *Evolution impact from defect density:* A decreasing defect density indicates that fewer corrections are gradually needed (as the code matures), and thereby a higher quality level is achieved for these reusable components individually. The findings support the results from literature.

- *Evolution impact from change density:* A decreasing change density for the reusable components could indicate that they become more mature as they are being adapted to accommodate new and altered requirements in relation to new and existing "client" software components. It could also be a natural result of improved reliability due to operational testing through usage (which is inherently higher for the reusable components).

5.3 RQ3: What are the impacts of Test Driven Development versus test-last development on reusable components?

Contribution C3: Improved understanding of the impact and effectiveness of TDD; article P4

- *Impact on defect density:* There is a reduction in mean **defect density** for TDD compared to traditional development methodology. The discovery of more new defects means that additional test cases (or validation/verification cases) are included into the test suites over several releases. In this way, the developed tests in TDD remain a valuable asset towards reuse. The refactoring practices inherent to TDD also aid in this direction.

- *Impact on change density:* There is a higher mean **change density** for TDD compared to traditional development methods, indicating that the reusable components are more adaptable (rather than having lower quality). In this regard, an underlying well-defined architecture allows a high level of modification. The introduction of new requirements from other systems (as the reusable components are being adopted by new divisions and departments), along with context factors such as prior knowledge of and experience with TDD, also plays a role here.

- *Additional factors:*
    - **Context:** The inherent characteristics of reusable components include:
        - potentially higher change density,
        - increasing abstraction level,
        - higher number of effective users and middleware-like position,
        - stable application domain and APIs.

      These should be considered as part of the context for writing new test cases.

    - **Refactoring:** The associated overhead is a worthwhile investment towards future reuse and adaptability.

    - **Lack of Design:** Shortcomings with respect to design for reusable components can be handled by implementing and using additional design and documentation practices.

5.4 RQ4: What are the perceived architectural risks of CBSE-driven software evolution, and how can these risks be mitigated?

Contribution C4a: Identification of perceived risks and related mitigation strategies specifically for the evolution of software architecture; articles P5, P6.
Contribution C4b: An adapted operational matrix for risk management in software architecture evolution; article P6

- *Identification of the most influential architectural risks:* We have explored planning and development risks in projects, on the technical, process and organizational levels. These risks indicate that while some efforts towards proper risk management have already been made, further improvements are warranted in terms of learning and reflection. Furthermore, the identified risks span many different issues, showing that architectural risk management in software evolution must consider a wide range of factors.

- *Identification of the most successful risk mitigation strategies:* The risk mitigation strategies were also identified according to technical, process and organizational levels, as matched to the identified risks. The low complexity level of some of these strategies indicates that risk management adoption does not necessarily require a large investment up front.

- *An operational matrix tool for risk management in software architecture evolution:* This tool is intended for use in risk management within software projects, where evolution of the software architecture is an issue in one or more forms. Existing perceived risks are matched with corresponding strategies, which can be used "as is" or as basis for further elaboration.

5.5 Overall summarized Discussion of research results

Our research aims to investigate the impact of modern trends in CBSE regarding the development process, defects and changes to individual reusable software components, and risk management of software architecture evolution.

Impacts of evolution on development processes are different for in-house development vs. OTS-based development, as outlined in Chapter 4. Nevertheless, rather than using radically different lifecycle processes when reusing OTS components, developers utilize and extend established development practices to accommodate specific aspects of external components. Possible explanations for this include:

- *Cost factors:* To limit costs (i.e. budget and schedule) of software development remains important. Building on already established processes seems to support this focus.
- *Convenience and stability:* It appears that chosen development processes in a company can be revised sufficiently for adoption and evolution of OTS components.

Our investigation further focused on the evolution impact on individual reusable software components in terms of defect and change densities. Lower defect densities were shown for these components over several releases, while for five of six components decreasing change densities were shown over several releases. A drop in defect density was also shown after the introduction of TDD on the development of the reusable components. Influencing factors include:

- *Broader range of impacts:* The reusable components are intended for use with many other (reusable as well as non-reusable) software components, and have a higher number of potentially diverse requirements to accommodate.
- *Higher inherent maturity of the reusable components*: Changes to the reusable JEF framework appear to become fewer and less complex as the framework matures, and hence becomes better suited to the various systems that it must serve. This indicates that the reusable components reflect well-considered abstractions. This also indicates that the growth in reliability due to operational testing helps in further enhancing the maturity of reusable components.
- *Process impact on components for TDD:* The tests developed with TDD remain a valuable asset towards reuse, as are the refactoring practices inherent to TDD (as indicated by the reduction in the average defect density for the investigated components). A lower change density is desired towards component maturity, while the higher change density shown for development of reusable components with TDD indicates that a higher level of adaptation (i.e. the amount of new and altered requirements) was incurred here.

Perceived risks and corresponding risk management strategies in software architecture evolution are important, as they provide a starting point for further improvement of relevant practices. They also enable structured process adaptations towards an ultimate goal of improved software quality. Important points include:

- *Planned vs. encountered risks:* The larger number of identified risks appears during project planning, rather than being encountered later during evolution. Thus, there is already a basis for implementation of improved risk management practices.
- *Risk management strategies currently in use:* Defined and documented evaluation of software architectures enables system architects to discover design errors and conflicting requirements early in the process, potentially saving a project from more significant problems later. In this thesis investigation, we find risks that mirror this concern, such as PR 2 (Table 4). Nevertheless, only about 21% of the respondents indicate this risk's influence as "Very High" or "High". Also, the corresponding mitigation strategies we identified (Table 4: PS 1, PS 3, PS 4) merely express recovery from missing evaluation output.

5.6 Discussion of Contributions in relation to State-of-the-art

The impact of introducing software reuse (in-house or COTS/OSS), from the viewpoint of developers working with software integration, was studied in articles **P1** and **P2**. In article **P1**, the key positive impacts found match those described in the literature [Lim 1994]. Reuse training was shown to be useful towards facilitating reuse,

which also supports earlier work [Frakes 1995]. Our results on developers' understanding of reusable components also support the findings from [Li 2004].

The investigation on OTS components in article **P2** was initially inspired by an investigation on the usage of COTS components performed by Torchiano et al. [Torchiano 2004]. They proposed six new "theses" that challenged previous research. In our investigation, we were able to support four of the six new "theses". The unsupported theses were:

- "standard mismatches were more frequent than architecture mismatches";
- "OTS components were mainly selected based on architecture compliance instead of function completeness" (see article SP4).

Our findings on OTS were further elaborated to provide the ten "facts" in **P2**. Among these facts, related work on risks and risk mitigation in OSS emphasizes the importance of the relationship between the developing organization and the component provider [Hauge 2010]. Also, component selection is commonly constrained by project-specific properties [Hauge 2009a]; developers often select the first working component instead of evaluating options and then making a selection based on the best fit.

With respect to our research on defect and change density in article **P3**, a later related study investigated the overall combined defect density of reusable components vs. client applications (i.e. complete combined releases) over their respective current lifetimes (i.e. releases thus far) [Gupta 2009a]. They found that the overall defect density of the reusable components was lower than that of one application and marginally higher than that of another application, partly contributing this latter "imbalance" to defects due to poorly implemented functionality parts.

In article **P4**, neither software evolution nor reusable components appear to have been explicitly investigated in earlier work on Test Driven Development. The reduction in mean defect density we found (36%) is similar to that found for non-reusable components in prior studies [George 2004] [Maximilien 2003] [Janzen 2005]. Also, non-corrective changes do not appear to have been explicitly considered in earlier work on TDD.

George et al. [George 1994] indicate that context (e.g. TDD training) exerts an important influence on writing new test cases. Our results indicate that characteristics of reusable components (e.g. potentially higher change density, increasing abstraction level, higher number of effective users and middleware-like position) should be part of this context consideration when developing such components. The refactoring inherent in TDD is also seen as a disadvantage due to the added overhead [George 1994].

However, our results indicate that for reusable components this overhead may be worthwhile regarding future reuse and adaptability. We also did not see any indication that lack of design was a problem, though this was reported for non-reusable components in [George 1994].

In articles **P5** and **P6**, in relation to Boehm's framework [Boehm 1991], we investigated issues related to identification, analysis, prioritization, assessment, planning and resolution of risks and strategies towards software architecture evolution. The 15 most influential risks (of 21) in our pilot survey fit into corresponding risk categories identified in work by Bass et al. [Bass 2007] (Table 9) and Ropponen et al. [Ropponen 2000] (Table 8). The same holds for 16 out of the 19 most influential risks in article **P6** (Table 10). This clearly shows the industrial relevance of our initial results. For Tables 8-10, duplicate identifiers are marked as #x, where x is the sequence

number. Also, while the identifiers in Tables 8 and 9 refer to article **P5**, the identifiers in Table 10 can be found both in Tables 3-5 as well as in article **P6**. We have also identified corresponding mitigation strategies (that appear not to have been investigated earlier) towards handling these risks.

Table 8: Relations between risk categories in the pilot risk survey (P5) and Ropponen et al. [Ropponen 2000]

ID	Ropponen et al. [Ropponen 2000]
	Requirements risks:
PR 4	"Insufficient requirements negotiation contributed to requirement incompatibilities"
TR 3	"Increased focus on modifiability contributed negatively towards system performance"
	Architecture Team risks:
OR 5	"Separate architecture team per maintenance/evolution cycle contributed to insufficient knowledge about the existing architectural design"
OR 7	"Large architecture team affected division of duties and subsequently implementation of maintenance/evolution cycle negatively"
OR 8	"Lack of clear lead architect affected implementation progress negatively and contributed to extra effort needed"
	Stakeholder risks (from the subcontractor viewpoint):
PR 3	"Lack of stakeholder communication affected implementation of maintenance/evolution cycle negatively"
OR 2	"Cooperative maintenance/evolution with architects from customer organization required extra training and communication efforts"
OR 3	"Lack of clear point of contact from customer organization contributed to inconsistencies in communication of the architecture and requirements"
PR 8	"Customer architects being unfamiliar with architecture change process affected maintenance/evolution cycle schedule negatively"

Table 9: Relations between risk categories in the pilot risk survey (P5) and Bass et al. [Bass 2007]

ID	Bass et al. [Bass 2007]
	Quality Attribute risk:
TR 3 (#2)	"Increased focus on modifiability contributed negatively towards system performance"
	Integration risks:
TR 4	"Varying release cycles for COTS/OSS components made it difficult to implement required changes"
OR 4	"Not allowed to change OSS as decision mandate external to architecture team, affecting performance and modifiability negatively"
	Requirements risks:
PR 4 (#2)	"Insufficient requirements negotiation contributed to requirement incompatibilities on the architecture"
TR 3 (#3)	"Increased focus on modifiability contributed negatively towards system performance"
	Documentation risks:
PR 1	"Lack of architecture documentation contributed to more effort being used on planning the maintenance/evolution"
PR 6	"Using Software Change Management system without explicit software architecture description contributed to inaccuracies in communicating the architecture"
	Process and Tools risks:
PR 2	"Lack of architecture evaluation delayed important maintenance/evolution decisions"
PR 6 (#2)	"Using Software Change Management system without explicit software architecture description contributed to inaccuracies in communicating the architecture"
	Allocation risks:
TR 1	"Poor clustering of functionality affected performance negatively"
TR 4 (#2)	"Varying release cycles for COTS/OSS components made it difficult to implement required changes"
	Coordination risks:
PR 3	"Lack of stakeholder communication affected implementation of maintenance/evolution cycle negatively"
PR 8	"Customer architects being unfamiliar with architecture change process affected maintenance/evolution cycle schedule negatively"
OR 2	"Cooperative maintenance/evolution with architects from customer organization required extra training and communication efforts"
OR 3	"Lack of clear point of contact from customer organization contributed to inconsistencies in communication of the architecture and requirements"
OR 4	"Not allowed to change OSS as decision mandate external to architecture team, affecting performance and modifiability negatively"

Table 10: Relations between risk categories in the main risk survey (article P6), Ropponen et al. [Ropponen 2000] and Bass et al. [Bass 2007]

Ropponen et al. [Ropponen 2000]	**Technical risks (TR)**	**Process risks (PR)**	**Organizational risks (OR)**
Requirements risks:	TR 2, TR 3, TR 6,	PR 4	
Architecture Team risks:			OR 2
Stakeholder risks (from the subcontractor viewpoint):		PR 5	OR 3, OR 4
Bass et al. [Bass 2007]			
Quality Attribute risk:	TR 6 (#2)		
Integration risks:	TR 5, TR 8	PR 6	OR 1
Requirements risks:	TR 3 (#2)	PR 4 (#2)	
Documentation risks:		PR 1	
Process and Tools risks:		PR 2	
Allocation risks:	TR 1, TR 7, TR 8 (#2)		
Coordination risks:		PR 5 (#2)	OR 1 (#2), OR 2 (#2), OR 3 (#2), OR 4 (#2)

5.7 General Recommendations to Practitioners

In terms of recommendations for general practitioners of software engineering, including those at StatoilHydro ASA, we would like to highlight the following:

- *Improvements of components and processes for handling of software evolution:*
  - *Standardized architecture:* In our results, a defined / standardized software architecture is seen as beneficial. Such an architecture should be evolved to accommodate future changes. An explicitly defined / standardized architecture could provide a proper basis for making such wide-reaching changes.

  - *Integration of management tools:* It is imperative that the project management tools, used for internal reporting in a company, are properly integrated with each other to enable proper data collection and analysis. In our case, Rational ClearCase (for SCM) and Rational ClearQuest (for handling TRs and CRs) tools were used for version control and defect/change reporting, respectively. Even though these two tools came from the same manufacturer, there was a lack of tool integration that made it difficult to obtain direct information on defects and changes (e.g. their "size") incurred on specific versions. Rather, the company had to rely on complementary sources to provide this information.

- *Reporting efforts:* There has to be commitment in the development organization to properly report data (defects, changes, effort, etc.) being collected for later analysis in the company. Our experience is that unless such data are directly related to their everyday work tasks, many developers see their reporting as unnecessary and even a target for cheating. Even the name of an affected component or effort usage (simplified just as small, medium, large) is frequently missing. The same "state-of-affairs" has been seen in dozens of companies according to [Mohagheghi 2006], thus enabling large changes may prove very difficult. Nevertheless, developers' input is crucial in providing relevant data for proper analysis towards product and process improvement.

- *Reuse training programs:* Our results indicate that although company reuse programs do exist, knowledge of these programs may be variable among developers. Organization and scheduling of such programs and related activities should hence be improved, especially to account for the needs of external personnel that may be added to a given project during its lifetime.

- *Knowledge sharing:* Rapid changes of personnel, requirements and resources can affect the quality specifications and related knowledge of reusable components negatively, as indicated in our results. It is therefore important that practitioners maintain knowledge sharing, regarding tacit and explicit knowledge, with their peers, and participate in the use and implementation of relevant collaboration tools (concrete examples include Microsoft SharePoint [Microsoft 2010] and open source suites such as Trac [Trac 2010]).

- *Improvements in risk management of software architecture:*
  - *Software architecture evaluation:* Software architecture evaluation should be implemented more explicitly as a complete end-to-end process. As aforementioned, the current focus is merely on recovering artifacts, rather than hindsight reflection and learning.

  - *Risk management strategies:* The median outcome rating for the strategies from our results for all three risk categories (technical, process, and organizational) was "Medium". So, there is still need for improvements in implementing risk management.

  - *Risk mitigation and training:* The focus of system architects' mitigation efforts appears currently to be on recovering needed architecture details and improving communication, while producing the system according to specification. Effort should therefore be made towards improving regular documentation and evaluation of the architecture, integrated with the maintenance / evolution process. Proper training of both architects and organizational management are means to achieve these improvements.

5.8 Relationships between contributions and overall SEVO goals

The relationships between the contributions in this thesis and the overall SEVO goals are as follows:

G1. Better understanding of software evolution, focusing on CBSE. We claim that this thesis advances the state-of-the-art within the field of software engineering, specifically in the context of the contributions **C1** and **C2**. These contributions address process and component aspects towards a profiling of software evolution.

G2. Better methods to predict the risks, costs and profile of software evolution in CBSE. Contribution **C3** addresses the effectiveness of Test Driven Development as a strategy to manage software evolution impact in terms of defects and changes. Contributions **C4a** and **C4b** specifically address the risk aspect of software evolution, investigating perceived risks and corresponding risk management strategies. Through these contributions, we have achieved a better understanding of impact in terms of perceived risks and management strategies related to software architecture evolution.

G3. Contributing to a national competence based around these themes. The work reported in this thesis has been published in refereed international conferences and journals (totaling 40 articles). All publications and related results have been reported to the FRIDA national database of research results. SEVO results have also been integrated in NTNU courses.

G4. Disseminating and exchanging the knowledge gained. We have established regular contact, and have several joint future publications, with other researchers with similar research interests. We have also presented our work at international conferences (e.g. ISESE, ICSEA), and arranged workshops (e.g. at Simula Labs in Oslo) to disseminate and further build on the knowledge we have gained through our investigations.

5.9 Reflections on research context: the role of our main industrial partner StatoilHydro ASA, and the software industry

An up-to-date research agenda is the cornerstone of any software engineering research project. Such a research agenda is best identified and further investigated through studying actual products and processes. This is best done in an industrial setting, constituting an "in-vivo" environment for software engineers, their work processes and produced software. Unfortunately, because of the nature of software and a dynamic marketplace, foci and activities are commonly diverse and vary widely between academia and industry. While academic researchers tend to favor a more long-lived perspective, industry is often more concerned with getting the right product to the right market at the right time. In this respect, we are very thankful to have had a stable, patient, large industrial partner, namely StatoilHydro ASA, which was willing to let us study their systems for almost four years towards some of the investigations in this thesis.

When trying to involve ourselves as academic researchers in industrial studies, it is important to regularly show industry short-term benefits of the proposed investigations. All the while we should still keep our own research agendas in mind and allow them to be as modifiable as possible when looking for new or altered opportunities. That is to say, we should *"have the serenity to accept the things we cannot change, the courage to change the things we can, and the wisdom to know the difference"* [Niebuhr 1934].

In our case, we were fortunate to have strong industrial contacts who partially shared research interests similar to ours, and who were willing to see them in light of a longer perspective. This made it possible to carry out industrial case studies as reported in this thesis, with results interesting both to us as researchers, and for the company. We also commonly got additional feedback upon informal requests. Thus, our research questions were formulated both through literary review and by inputs from developers and data collected at the company. Also, upper management was involved in giving feedback prior to article submissions to conferences, showing the company's level of interest in, and the relevance of, our research. It also "helped" that the main StatoilHydro IT manager and principal contact (H. Rønneberg) had a PhD from NTNU and a position as an adjunct associate professor at NTNU, and was acquainted with Dr. R. Conradi.

As mentioned, our experience is that missing, incomplete or inconsistent data is quite common at most companies [Mohagheghi 2006], and this was also found at StatoilHydro ASA. This indicates the lack of prior data analysis of available data, pointing towards a lack of systematic metrics, properly defined quality goals, and/or relevant resources towards this end.

It should also be mentioned that in parallel with these working contacts, we also had several attempts in establishing new contacts with other companies. These attempts proved much more difficult, and none worked out to yield relevant and long-term data for our research. This was also noted as an issue in carrying out the survey investigation in **P2**, where we involved respondents from companies in the European IT-industry. We can hence echo the comments made in **SP5** and related papers: That the IT-industry overall seems quite busy and with little time for joint research these days.

5.10 Threats to Validity in Software Engineering and towards the contributions of this thesis

An essential discussion regarding the study results relates to their validity. Empirical research in software engineering commonly draws on definitions of threats to validity that originate from the field of statistics. Not all the threats are relevant for all types of studies. Wohlin et al. [Wohlin 2000] define the following four categories of threats to validity (originally for experiments, but commonly used and found applicable for most types of empirical studies in software engineering):

- **Conclusion validity:** Concerning the "correct analysis", i.e. the relationship between the treatment or independent variable(s), and the outcome or dependent variable(s) in a study. A central question is whether the results are statistically significant in terms of statistical tests, considering p-value, statistical power and sample size.

- **Internal validity:** Concerning the data correctness, i.e. whether there is a causal relationship between the treatment and the outcome. The main threat is possible effects caused by factors not explicitly considered, e.g. unknown bias, and reporting wrong or misunderstood data.
- **Construct validity:** Concerning the "correct or relevant metrics", i.e. that the design of the study is correctly constructed to reveal something about the relationship(s) being studied. Threats here include mono-operation bias (i.e. single study program not reflecting the constructs), and mono-method bias (i.e. single measure type may be misleading).
- **External validity:** Concerning the "correct context", i.e. generalizability of results outside the study scope. The main threats here include having non-representative subjects, location or time in carrying out the study.

In this context, a treatment is the (collective) methods that the study object(s) are subjected to so that measurements can be made. In a simplified way, we can say that the outcome is the result produced, and obtained through these measurements.

Different threats have different priorities based on the research method. For example, when testing a formal theory, internal validity is most important. One major issue affecting validity in software engineering is that there is much "unused" data of poor quality. Yin et al. [Yin 2003] outline three tactics for improving the validity in case studies:

- Construct validity: Employ multiple data collection sources (triangulation) and use knowledgeable persons to review the report during composition.
- Internal validity: Employ matching of patterns (i.e. between empirically-based patterns and predicted patterns, in particular when it comes to explanatory studies), and incorporate competing explanations in the data analysis.
- External validity: Use theory towards research design when it comes to single case studies.

Validity determines the trust that can be placed in the obtained results, and thus shows the value of an investigation. Threats to validity for all the studies in this thesis are discussed below, and are further explained in each article in Appendix A.

Validity of the industrial case studies (P3, P4):

- **Construct validity:** The analysis constructs we have used (defect density, change density) are based on well-founded concepts in the software evolution field. Our research questions similarly have their basis in the research literature. All our data on change requests and trouble reports is pre-delivery, from the development phases of each new subsequent release. Also, all our data is based on complete and stable releases, and we have used all the releases available at the time of the individual studies.

- **External validity:** The data set for each of the two industrial case studies is taken from one single company, StatoilHydro. The case studies are of individual industrial systems, an issue that nevertheless remains a threat. Our results should be relevant and valid for other releases of these components. Generalization to

similar contexts in other organizations should be discussed on a case-by-case basis.

- **Internal validity:** All of the data has been extracted directly from StatoilHydro by us. Incorrect or missing data details may exist, but those records with missing details that are related to our analysis have been excluded. All of the data comes from complete and stable releases. We have performed the analysis in a cooperative manner, allowing cross-checking to ensure compliance, consensus and correctness. Also, the tools used for data extraction and data analysis (Rational ClearQuest, SPSS and Excel) are well-known. The actual defect and change reporting procedures nevertheless represent a threat (although we have made an effort to exclude any invalid or incomplete data), since there may be uncertainty related to whether a particular issue is classified through a trouble report or change request. Another threat is the close interaction between the reusable and non-reusable components that may have led to issues being incorrectly assigned to the different system parts (non-reusable DCF vs. reusable JEF).

- **Conclusion validity:** The analyses are based on relatively small data sets. We did collect complete sets of data, which thus should be sufficient to draw relevant and valid conclusions. Confounding factors, such as differences in developer experience between teams, can represent threats, but since the studied systems were developed within the same organizational unit we do not consider this a threat in our studies. Roughly 1/3 of the developers worked across the systems, while the remaining personnel had common experience, skill and educational levels.

Validity of the surveys (P1, P2, P5, and P6):

- **Construct validity:** Our research questions are firmly rooted in the research literature, and the actual questions in the questionnaires and interview guides are directly related to the research questions. The research and questionnaire / interview guide questions were further adapted towards our use, and pre-tested among local colleagues and industrial panels to allow refinement. All terminology used has been defined in the questionnaires / interview guides to provide clear definitions and avoid misinterpretations.

- **External validity:** The fact that three of our surveys use variants of convenience sampling **(P1, P5)** and constrained snowball sampling **(P6)** presents a threat. It should be noted that obtaining a random sample is almost unachievable in software engineering studies due to the lack of reliable and comprehensive demographic background data about the relevant populations of projects and companies. Nevertheless, we managed to use stratified-random sampling in our largest survey **(P2)** encompassing the IT-industries in Norway, Germany and Italy, at the cost of circa two person-weeks per filled-in questionnaire [Conradi 2005] (mostly caused by unknown gatekeepers). We also ensured that the respondents had relevant background and experience. All of the respondents of

the three smaller surveys **P1, P5** and **P6** are nevertheless from the Norwegian software-intensive IT-industry, an issue that remains a limitation.

- **Internal validity:** The respondents were all well-qualified professionals from the software-intensive IT-industry, and had expressed a definite interest in the study. All of them had the required knowledge and background to provide relevant answers. We therefore believe that they answered the given questions to the best of their ability, truthfully, and drawing on their own experiences and knowledge. We also clarified any ambiguities in the questions and the accompanying definitions during actual interviews, in addition to the definitions provided in the questionnaires / interview guides themselves. Minor issues that arose in the larger survey **(P2)** were a few linguistic errors in the actual questionnaire, and in the translation of the questionnaire from English to the national languages. Overall though, the questionnaire proved to be very robust in this regard.

- **Conclusion validity:** The relatively small sample size in **P1** and **P5** remains a threat, but still yields interesting and valuable insights. In **P5** we initially identified several issues that may constitute architectural risks for evolving systems. These insights also functioned as a background for refining the interview guide for an expanded sampling base for the larger survey in **P6**. In summary, this means that the larger survey **(P6)** relied heavily on the findings from the survey in **P5**.

6 *Conclusion*

The research in this thesis investigates and reports on the impact of modern trends in CBSE on software processes. It also investigates the impact of CBSE-driven software evolution (as defect density and change density) on individual reusable components. Finally, it investigates risk and risk management related to the CBSE-driven evolution of software architecture. Our research has yielded valuable insights and resulted in four contributions. This last chapter sums up our findings and recommendations, and also outlines possible directions for future work.

6.1 Overall Summary of Findings

We first revisit the main investigation themes of this thesis as a prelude:

- the influence of modern trends in CBSE on software processes (since there is a lack of large-scale empirical studies on the adoption of new processes, to study their characteristics and impacts),
- evolution and its impact on CBSE (as an empirical basis for more targeted handling of software evolution), and
- perceived architectural risks and corresponding risk mitigation strategies in CBSE-driven software evolution (to support improved systematic handling of risks and mitigation strategies in software architecture evolution).

Findings:

- The **focus in response to the impact of modern trends in CBSE on the development process** was found to be **on keeping budgets and schedules**, which is important in most software organizations. This is a reason why new concerns around CBSE-driven evolution are met by integrating light-weight changes into already existing company processes, rather than implementing new alternative processes from scratch. The ability to efficiently adopt these new mechanisms is also important.
- **CBSE-driven evolution impacts reusable software components** in terms of a never-stopping **stream of potentially diverging requirements** from various "clients" (i.e. non-reusable components). These different types of components will thus be impacted in different ways, also since reusable software components are used by non-reusable components in new development.

- **Reusable components have a higher maturity,** due to being based on well-considered abstractions as well as the perpetual fixing of post-delivery defects and changes in subsequent new releases.
- Our results on TDD of reusable components show that the **test cases and inherent refactoring in TDD are beneficial for software reuse**, as well as the **ability** of the reusable components **to adapt** to new contexts and requirements.
- On **actual perceived architectural risks in CBSE**, we found that the **majority** of identified risks were **included in planning**, with a smaller number being encountered later when the actual changes were implemented.
- **Software architecture evaluation**, which is an effective technique towards discovering architectural design problems and conflicts early, **is commonly only partially employed** (i.e. the focus is only on attaining the evaluation output rather than properly implementing the full process).

Finally, we have also used our results to propose improvements towards both general practitioners and practitioners at StatoilHydro ASA (Chapter 5).

6.2 Recommendations for CBSE Researchers

We have provided an updated definition of software evolution [P5], building on the one found in [Mohagheghi & Conradi 2004a]. This definition will aid in facilitating further studies of software evolution, as it specifically includes both code and other software artifacts, as well as quality aspects of software architecture evolution.

The table-driven tool we have proposed for effective handling of architectural risks in software evolution (Tables 3, 4, and 5) can serve as a base to incorporate future insight on architectural risks in software evolution.

The two metrics used, defect density and change density, are well-known in the literature as measures of software evolution [Mohagheghi 2004a] [Mohagheghi & Conradi 2004b]. However, these metrics give only a limited view of software evolution, without taking aspects such as component complexity into account. We would therefore encourage researchers to work towards improved and more detailed metrics for assessing component reliability and modifiability in software evolution.

6.3 Recommendations for Practitioners regarding Modern Trends in CBSE

Our recommendations to general practitioners are also applicable for practitioners at StatoilHydro ASA, and vice versa. Our results show that improvements can be made as follows:

- *A standardized architecture* in the company should function as a platform for adapting the JEF framework of reusable components towards new requirements and further evolution at StatoilHydro ASA. To follow up such a goal, the company should:
    - Keep the architectural design and related assets as ‘live’ artifacts, allowing updates dynamically as the architecture is evolved.

    - Ensure and maintain long-lasting commitment to empirical studies at all levels of the developing organization. Developers in our context are inherently skeptical towards tasks (read: "extra work") that could not explicitly be related to their daily work.

- *Knowledge sharing* to counter frequent changes of
    - personnel,
    - requirements, and
    - resources

  can affect the combined collective knowledge of a team or even an entire software development organization. This means that maintaining and encouraging continuous knowledge sharing and exchange is important, in order to allow the organization to retain relevant knowledge once a developer leaves. A related recommendation appears in [Chen 2007], where the authors urged organizations to appoint a "component uncle" to follow up the evolution of each adopted OSS component, or a component in a certain area or from (a) certain provider(s).

- *Training* can, depending on an individual's prior knowledge and experience, be an important precursor to effective knowledge sharing. Our results show that proper knowledge of a reuse training program was not present among all developers and hired-in consultants at StatoilHydro ASA. It is therefore important that such programs be properly promoted and carried out.

- *Test Driven Development* can help promote reuse due to the higher inherent focus on testing and refactoring. While our results show that TDD can help reduce the number of defects in comparison to test-last development for reusable components, effort measurements should also be a part of introducing TDD. This is because lower productivity has been shown in other industrial studies [Janzen 2005], albeit for non-reusable components.

- *Risk management*: system architects are currently focusing on obtaining and recovering evaluation artifacts and improving communication. At the same time, our results show that the median of the outcome ratings for the strategies was "Medium". We thus need to improve architectural documentation artifacts, and evaluate the architecture on a regular basis. Furthermore, these aspects should be integrated seamlessly into the evolution process. Implementing a training program that covers both architects and organizational management seems to be a sensible way to achieve this.

6.4 Future Work

The described case studies provide a basis for further studies in Component-Based Software Engineering, software evolution and software architecture as follows:

With respect to the impact of defects and changes in software evolution (and related to SEVO goal G1):

- *A study of cost estimation related to OTS software:* StatoilHydro uses a large amount of OTS software, requiring a certain amount of integration and tailoring for such components. Personal experience as the basis for estimating integration effort appears to be common in industry, and although it often leads to inaccurate estimates [**P2**], expert estimates are usually better than formal-model ones. The question remains how to define who is an expert. StatoilHydro aims to determine the total costs associated with integration and tailoring, as well as their distribution over change types for OTS components. Additionally, to investigate possible differences with respect to reuse of in-house components is of interest.

- *A study on complexity:* Data on defect density and change density [Mohagheghi 2004a] [Mohagheghi & Conradi 2004b] [Mohagheghi & Conradi 2008] is commonly used towards investigating reliability and maintainability of software components. Nevertheless, these metrics do not include any weighting of complexity of individual components. Function points is a method that has been proposed to include software complexity in the size measure [Umholtz 1994], but as noted, no standard method for counting function points that includes algorithmic complexity is currently supported by ISO [ISO 2010]. This would be a study towards improved metrics for providing a more detailed view of component quality and reliability in software evolution.

With respect to the impact of modern trends in CBSE on the development process (and related to SEVO goal G2):

- *A study of knowledge sharing:* Certain teams at StatoilHydro ASA (indeed in most ICT companies) undergo fast changes with respect to personnel, requirements and resources [Gupta 2009a]. The company is concerned with how to best maintain knowledge which would otherwise be lost. There is some knowledge sharing technology in place, but this may not fulfill the needs of the company. Further studies are necessary to investigate the potential role of collaboration tools (e.g. Wikis, networks, ecosystem) in the company.

- *COTS/OSS and communities:* Earlier investigations of OSS and communities have focused on the current state-of-practice within industry with respect to component selection and integration [Ayala 2007] [Scacchi 2006b]. Future research on current best practices within OSS selection should have a more detailed focus on evaluating benefits versus disadvantages to get the full picture about efficiency, and allow researchers to propose relevant improvements where needed.

- *Improvements to a company's software processes:* We have made several recommendations for process improvements towards general practitioners and at

StatoilHydro ASA (see Chapter 5). Additional investigations are needed to study the implementation and effects of these recommendations.

- *Agile Architecture:* As noted in Chapter 2.4, the long-term perspectives involved in software architecture and short-term foci of agile software development should be combined to allow the benefits of both to prevail. Additional studies are necessary to explore how these two paradigms can best be combined.

- *Further study of the approach to TDD at StatoilHydro:* The company aims to investigate the effort used towards handling defects and changes, and to investigate other potential facets of benefits related to TDD.

With respect to architectural risks in software evolution (and related to SEVO goal G2):

- *Study of code-level and other artifact data for software architecture evolution:* We want to couple the risks and corresponding risk management strategies we have identified with an investigation of empirical data related to architecture evolution. This can, in the future, facilitate a method framework for better handling of these issues, also on the code-level.

- *Expand our knowledge on architecture risk management in software evolution:* Though several possible concepts and related activities towards effective risk management in CBSE have been proposed, there is a lack of actual empirical studies in the area [Glass 2001]. This means that the actual value and effectiveness of proposed activities and tools remain largely unknown. Our studies represent a start to investigate certain software architecture risk issues in connection with software evolution. We would nevertheless like to expand our survey base to possibly confirm and/or add to our findings thus far. Performing hands-on investigations (e.g. case studies) of actual evolution of software architecture also remains a priority issue.

- *Thorough investigation of software architecture evaluation methods:* Earlier investigations on software architecture analysis [Bass 2007][O'Connell 2006] have focused on structured analysis outputs as a basis for determining risks. However, the actual methods used for analyzing the software architecture can vary quite a lot [Babar 2007a]. Investigating a wider range of analysis methods will help to discover risk issues possibly missed by earlier studies.

- *A study of architectural versus non-architectural changes:* Failure of the software architecture can cause failure of an entire development project. Better knowledge and understanding about architectural evolution can potentially improve handling of actual software architecture changes, which can cause subsequent changes in many components of a software system [Bass 2004]. Such an investigation should study architectural versus non-architectural changes, with respect to their distribution and possible handling.

Glossary

Change density: Number of Change Requests (CRs - perfective, adaptive and preventive (but not corrective) changes) per Non-commented Source Lines of Code [Mohagheghi & Conradi 2004a, p. 1] (defined as "change-proneness" in the reference).

Component-Based Software Engineering: *The process of defining, implementing, and integrating or composing loosely coupled, independent components into software systems* [Sommerville 2010, p. 453].

Defect density: Number of corrective changes (defects) per Non-commented Source Lines of Code [Mohagheghi & Conradi 2004b, p. 1].

Software architecture: *defined as the structure(s) of the system, which comprise(s) software elements, the externally visible properties of those elements, and the relationships among them* [Bass 2004, p. 21].

Software evolution: *the systematic and dynamic updating in new/current development or reengineering from past development of component(s) (source code) or other artifact(s) to accommodate new functionality, improve the existing functionality, or enhance the performance or other quality attribute(s) of such artifact(s) between different releases* [P5, p. 3].

Software maintenance: *the general process of changing a system after delivery* [Sommerville 2010, p. 242].

Software Process Improvement: *encompasses the understanding and changing of existing processes to improve software product quality, as well as to reduce costs and development time* [Sommerville 2010, p. 706, adapted].

Software reuse: the systematic reuse of components and other artifacts (e.g. abstractions, objects, or even applications) [Sommerville 2010, p. 194].

Software Risk: An issue that can potentially result in an unsatisfactory outcome, if not handled correctly, can be considered a risk [Boehm 1991, p. 33].

References

[Abts 2000] C. Abts et al.: *COCOTS: A COTS Software Integration Cost Model - Model Overview and Preliminary Data Findings,* Proc. 11th ESCOM Conference, 2000, pp. 325- 333.

[Albrecht 1979] A. J. Albrecht: *Measuring Application Development Productivity*, Proc. Joint SHARE, GUIDE, and IBM Application Development Symposium, Monterey, California, October 14-17, 1979, IBM Corporation, pp. 83–92.

[Arisholm 2004] E. Arisholm, D. Sjøberg: Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software, *IEEE Trans. Software Engineering,* 30(8):521-534, July 2004.

[Ayala 2007] C. Ayala, C. Sørensen, R. Conradi, X. Franch, and J. Li, *Open Source Collaboration for Fostering Off-The-Shelf Components Selection,* in Joe Feller and Alberto Sillitti (Eds.) Proc. Third International Conference on Open Source Systems (OSS 2007), Limerick, Ireland, June 11-14, 2007, WG 2.13 Working Conference, Springer IFIP Series No. 234, pp. 17-30.

[Babar 2007a] M. Ali Babar, L. Bass and I. Gorton: *Factors Influencing Industrial Practices of Software Architecture Evaluation: An Empirical Investigation*, Proc. Quality of Software Architectures, Medford, Massachusetts, USA, July 12-13, 2007, pp. 90-107.

[Babar 2007b] M. A. Babar and B. Kitchenham: *The Impact of Group Size on Software Architecture Evaluation: A Controlled Experiment,* First Int'l Symposium on Empirical Software Engineering and Measurement, Madrid, Spain, 20-21 September 2007, pp. 420-429.

[Bannerman 2008] P. L. Bannerman: Risk and risk management in software projects: A reassessment, *J. Systems and Software,* 81(12):2118-2133, December 2008.

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

[Baskerville 1999] R. Baskerville and J. Pries-Heje: Knowledge capability and maturity in software management, *ACM SIGMIS Database,* 30(2):26-43, 1999.

[Bass 2001] L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau, *Volume I: Market Assessment of Component-based Software Engineering* in SEI Technical Report number CMU/SEI-2001-TN-007, 2001.

[Bass 2004] L. Bass, P. Clements and R. Kazman: *Software Architecture in Practice,* Second Edition, Addison-Wesley, 2004.

[Bass 2007] L. Bass, R. Nord, W. Wood and D. Zubrow: *Risk Themes Discovered Through Architecture Evaluations*, Proc. 6th Working IEEE/IFIP Conf. Software Architecture (WICSA), Mumbai, India, January 6-9, 2007, pp. 44-54.

[Basili 2008] V. R. Basili, D. Rombach, K. Schneider, B. Kitchenham, D. Pfahl and R. W. Selby (Eds.): *Empirical Software Engineering Issues,* Proc. Int'l. Workshop Dagstuhl, Germany, June 2006, Springer LNCS 4336.

[Bennett 2000] K. H. Bennett and V. Rajlich: *Software maintenance and evolution: a roadmap,* Int'l Conf. Software Engineering (ICSE) – Future of Software Engineering, Limerick, Ireland, June 4-11, 2000, pp. 73-87.

[Bertolino 2007] A. Bertolino: *Software Testing Research: Achievements, Challenges, Dreams*, Proc. Future of Software Engineering (FOSE'07) workshop, Int'l Conf. Software Engineering (ICSE), Minneapolis, MN, USA, May 23-25, 2007, pp. 85-103.

[Blundell 2005] C. Blundell, D. Giannakopoulou, C. S. Pasareanu: *Assume-guarantee testing*, Proc. 4th workshop on Specification and Verification of Component-Based Systems (SAVCBS'05), ESEC/ACM SIGSOFT Symp. Foundations of Software Engineering (FSE), ACM Press, 5-6 September 2005, pp. 7-14.

[Boehm 1988] B. W. Boehm: A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61-72, May 1988.

[Boehm 1989] B. Boehm: *Software Risk Management*, IEEE Press, 1989.

[Boehm 1991] B. W. Boehm, Software Risk management: Principles and Practices, *IEEE Software*, 8(1):32-41, January 1991.

[Boehm 1999] B. W. Boehm and C. Abts: COTS integration: Plug and Pray?, *IEEE Computer,* 32(1):135-138, January 1999.

[Booth 2004] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, "Web Services Architecture", W3C Working Group Note, 11 February 2004.

[Bouwers 2009] E. Bouwers, J. Visser and A. van Deursen: *Criteria for the Evaluation of Implemented Architectures*, Proc. 25th Int'l Conference on Software Maintenance, Edmonton, Canada, September 20-26, 2009, IEEE press, pp. 73-82.

[Breivold 2008] H. P. Breivold, I. Crnkovic, R. Land and M. Larsson: *Analyzing Software Evolvability of an Industrial Automation Control System: A Case Study,* Proc. Third Int'l Conference on Software Engineering Advances, Sliema, Malta, October 26-31, 2008, IEEE Computer Society, pp. 205-213.

[Buchgeber 2008] G. Buchgeber and R. Weinreich: *Integrated Software Architecture Management and Validation,* Proc. Third Int'l Conference on Software Engineering

Advances (ICSEA), Sliema, Malta, October 26-31, 2008, IEEE Computer Society, pp. 427-436.

[CACM's Inside Risks 2010] Communications of the ACM, Inside Risks Column, http://www.csl.sri.com/users/neumann/insiderisks.html, accessed 09 May 2010.

[Chen 2007] W. Chen, J. Li, J. Ma, R. Conradi, J. Ji, and C. Liu: *An Industrial Survey of Software Development with Open Source Components in Chinese IT Industry*, in Q. Wang, D. Pfahl and D. M. Raffo (Eds.): Proc. International Conference on Software Process (ICSP 2007 - in conjunction with ICSE'2007), Minneapolis, MN, USA, May 19-20, 2007, Springer Verlag LNCS 4470, pp. 208-220.

[Chen 2008] W. Chen, J. Li, J. Ma, R. Conradi, J. Ji, and C. Liu: An Empirical Study on Software Development with Open Source Components in Chinese Software Industry, *Software Process: Improvement and Practice (SPIP)*, 13(3):233-247, May/June 2008 (Upgraded from article at ICSP'2007, Minneapolis, 19-20 May, 2007).

[Chidamber 1994] S. R. Chidamber and C. F. Kemerer: A Metrics Suite for Object Oriented Design, *IEEE Trans. Software Eng.*, 20(6):476-493, June 1994.

[CMM] http://www.sei.cmu.edu/cmm/, 2010.

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

[Cooper 2008] D. R. Cooper and P. S. Schindler: *Business Research Methods*, 10th Edition, McGraw-Hill Press, International Edition, 2008.

[Conradi 2005] R. Conradi, J. Li, O. P. N. Slyngstad, C. Bunse, M. Torchiano and M. Morisio: *Reflections on conducting an international CBSE survey in ICT industry*, Proc. of the 4th International Symposium on Empirical Software Engineering (ISESE), Noosa Heads, Australia, November 17-18, 2005, IEEE Press, pp. 214-223.

[Conradi 2002] R. Conradi and A. Fuggetta: Improving Software Process Improvement, *IEEE Software*, 19(4):92-99, July/August 2002.

[Creswell 2003] J. W. Creswell: *Research Design, Qualitative, Quantitative and Mixed Method Approaches*, Sage Publications, London, UK, 2003.

[Crnkovic 2002] I. Crnkovic, B. Hnich, T. Jonsson and Z. Kiziltan: Specification, Implementation, and Deployment of Components, *Communications of the ACM*, 45(10):35–40, 2002.

[Crnkovic 2000] I. Crnkovic and M. Larsson: *A Case Study: Demands on Component-based Development*, Proc. 22nd Int'l Conf. Software Engineering (ICSE), Limerick, Ireland, June 4-11, 2000, pp. 21-31.

[Davison 2004] R. Davison, M. G. Martinsons, and N. Kock: Principles of canonical action research, *Information Systems Journal*, 14(1):65-86, 2004.

[Denzin 1994] N. K. Denzin and Y. S. Lincoln: *Handbook of Qualitative Research,* Sage Publications, London, UK, 1994.

[D'Ambros 2008] M. D'Ambros: *Supporting Software Evolution Analysis with Historical Dependencies and Defect Information,* Proc. 24th Int'l Conference on Software Maintenance, Beijing, China, 28 September 28 – October 4, 2008, IEEE press, pp. 412-415.

[Eisenhardt 1989] K. M. Eisenhardt: Building Theories from Case Study Research, *Academy of Management Review,* 14(4):532-550, 1989.

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

[Erdogmus 2005] H. Erdogmus, M. Morisio and M. Torchiano: On the effectiveness of test-first approach to programming, *IEEE Trans. Software Eng.,* 31(1):1–12, January 2005.

[Feller 2002] J. Feller and B. Fitzgerald: *Understanding Open Source Software Development*, Addison-Wesley, 2002.

[Finkelstein 2000] A. Finkelstein and J. Kramer: *Software Engineering: a Road Map,* Proc. 22nd Int'l Conf. Software Engineering, Future of Software Engineering Track, Limerick Ireland, June 4-11, 2000, pp. 3-22.

[Fitzgerald 2004] B. Fitzgerald: A Critical Look at Open Source, *IEEE Computer,* 37(7):92-94, July 2004.

[Fitzgerald 2006] B. Fitzgerald: The transformation of OSS, *MIS Quarterly,* 30(3):587-598, September 2006.

[Flyvbjerg 2006] B. Flyvbjerg: Five Misunderstandings About Case-Study Research, *Qualitiative Inquiry,* 12(2):219-245, April 2006.

[Foote 1997] B. Foote and J. Yoder: *Big ball of mud,* 4th Conference on Patterns, Languages of Programs, Monticello, Illinois, September 2-5, 1997, (Technical report #wucs-97-34, Dept. of Computer Science, Washington University Department of Computer Science, September 1997).

[Fowler 2001] F. J. Fowler, *Survey Research Methods (Applied Social Research Methods),* 3rd edition, Sage Publications, 2001.

[Frakes 1995] W. B. Frakes and C. J. Fox: 16 Questions on Software Reuse, *Communications of the ACM*, 38(6):75-87, June 1995.

[FSF 1985-2010] Free Software Foundation, www.fsf.org, accessed 18 May 2010.

[Fuggetta 2000] A. Fuggetta: *Software Process: A Roadmap,* Proc. 22nd International Conference on Software Engineering, Future of Software Engineering Track, June 4-11, 2000, Limerick Ireland, IEEE CS Press, pp. 25-34.

[Gemmer 1997] A. Gemmer: Risk Management: Moving Beyond Process, *IEEE Computer*, 30(5):33-41, May 1997.

[George 2004] B. George and L. Williams: A structured experiment of test-driven development, *Information and Software Technology*, 46(5):337–342, 2004.

[Glass 1994] R. L. Glass: The software-research crisis, *IEEE Software*, 11(6):42-47, November 1994.

[Glass 1999] R. L. Glass: The realities of software technology payoffs, *Communications of the ACM*, 42(2):74-79, 1999.

[Glass 2001] R. L. Glass: Frequently Forgotten Fundamental Facts about Software Engineering, *IEEE Software*, 18(3):110-112, May/June 2001.

[Glass 2004] R. L. Glass, V. Ramesh, and V. Iris: An Analysis of Research in Computing Disciplines, *Communications of the ACM*, 47(6): 89-94, 2004.

[Gotzhein 2006] R. Gotzhein and F. Khendek: *Compositional testing of communication systems,* Proc. 18th IFIP Int'l. Conf. Testing of Communicating Systems (TestCom), May 16-18, 2006, Springer LNCS 3964, pp. 227-244.

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

[Gupta 2010] A. A. Gupta, J. Li, R. Conradi, H. Rønneberg E. Landre: Change Profiles of a Reused Class Framework vs. two of its Applications, *Information and Software Technology*, 52(1):110-125, January 2010.

[Gupta 2009a] A. A. Gupta, J. Li, R. Conradi, H. Rønneberg, and E. Landre: A Case Study Comparing Defect Profiles of a Reused Framework and of Applications Reusing It, *Journal of Empirical Software Engineering*, 14(2):227-255, 2009.

[Gupta 2009b] A. A. Gupta: *The Profile of Software Changes in Reused vs. Non-Reused Industrial Software Systems*, PhD thesis, NTNU, 2009.

[Haller 2005] A. Haller, E. Cimpian, A. Mocan, E. Oren and C. Bussler: *WSMX - A Semantic Service-Oriented Architecture*, Proc. Int'l Conference on Web Services (ICWS), July 11-15, 2005, pp. 321-328.

[Hamlet 2006] D. Hamlet: *Subdomain testing of units and systems with state*, Proc. ACM/SIGSOFT Int'l. Symposium on Software Testing and Analysis, Portland, Maine, USA, July 17-20, 2006, ACM Press, pp. 85-96.

[Hamlet 2001] D. Hamlet and J. Maybee: *The Engineering of Software*, Addison-Wesley, Boston, 2001.

[Haney 1973] C. Haney, C. Banks, and P. Zimbardo: Interpersonal Dynamics in a Simulated Prison, *International Journal of Criminology and Penology*, 1(1):69-97, 1973.

[Hansen 2004] M. T. Hansen, N. Nohria and T. Tierney: What is your strategy for managing knowledge?, *Harvard Business Review*, 77(2):106-116, 1999.

[Hauge 2009a] Ø. Hauge, T. Østerlie, C.-F. Sørensen, and M. Gerea: *An Empirical Study on Selection of Open Source Software - Preliminary Results*, in A. Capiluppi and G. Robles (Eds.): Proc. 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS'2009), Vancouver, Canada, May 18, 2009, IEEE press, pp. 42-47.

[Hauge 2009b] Ø. Hauge and S. Ziemer: *Providing Commercial Open Source Software: Lessons Learned*, in C. Boldyreff, K. Crowston, B. Lundell, and A. I. Wasserman (Eds.): Proc. Fifth IFIP WG 2.13 International Conference on Open Source Systems (OSS'09) - Open Source Ecosystems: Diverse Communities Interacting, Skövde, Sweden, June 3-6, 2009, Springer Verlag, pp. 70-82.

[Hauge 2009c] Ø. Hauge, C.-F. Sørensen, and R. Conradi: *Adoption of Open Source in The Software Industry*, in B. Russo, E. Damiani, S. Hissam, B. Lundell, and G. Succi (Eds.): Proc. IFIP WG 2.13 Conference on Open Source - Development, Communities and Quality (OSS 2008), co-located with 20th World Computer Congress, Milan, Italia, September 7-10, 2008, Springer Verlag, pp. 211-221.

[Hauge 2010] Ø. Hauge, D. S. Cruzes, R. Conradi, K. S. Velle, and T. A. Skarpenes, *Risks and Risk Mitigation in Open Source Software Adoption: Bridging the Gap between Literature and Practice*, Proc. IFIP WG 2.13 6th International Conference on Open Source Systems (OSS'2010), Indiana, USA, 30 May - 2 June, 2010, Springer Verlag IFIP series, 15 pp.

[Hecht 2004] H. Hecht: *Systems Reliability and Failure Prevention*, Artech House Publishers, 2004

[ISO 2010] http://www.iso.org/iso/home.htm, accessed 18 May 2010.

[Janzen 2005] D. Janzen and H. Saiedian: Test-driven development: concepts, taxonomy and future directions, *IEEE Computer*, 38(9):43–50, September 2005.

[JEF 2006] JEF Concepts and Definition at Statoil ASA, http://intranet.statoil.com, 2006.

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

[Jones 2008] C. Jones: *Applied software measurement: Global Analysis of Productivity and Quality*, Third edition, McGraw-Hill, 2008.

[Järvensivu 2008] J. Järvensivu and T. Mikkonen: *Forging A Community Not: Experiences On Establishing An Open Source Project*, in B. Russo, E. DAmiani, S. A. Hissam, B. Lundell, G. Succi (eds.), Open Source Development Communities and Quality IFIP WG 2.13 on OSS, Milano, Italy, September 7-10, 2008, Springer Verlag, pp. 15-27.

[Kampenes 2009] V. B. Kampenes, T. Dybå, J. E. Hannay and D. I. K. Sjøberg: A systematic review of quasi-experiments in software engineering, *Journal of Information and Software Technology*, 51(1):71-82, 2009.

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

[Keil 1998] M. Keil, P. E. Kule, K. Lyytinen and R. C. Schmidt: A Framework for Identifying Software Project Risks, *Communications of the ACM*, 4(11):76-83, November 1998.

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

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

[Lehman 1985] M. M. Lehman and L. A. Belady: *Program Evolution – Processes of Software Change*, Academic Press, 1985.

[Lehman 1995] M. M. Lehman: *Process Improvement – The Way Forward*, Keynote Speech, Proc. 7th Int'l Conference on Advanced Information Systems Engineering (CAiSE), Jyväskylä, Finland, June 12-16, 1995, Springer LNCS 932, pp. 1-11.

[Lehman 1997] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry and W. M. Turski: *Metrics and Laws of Software Evolution – The Nineties View*, Proc. 4th Int. Symp. on Software Metrics (Metrics 97), Albuquerque, New Mexico, November 5-7, 1997, pp. 20-32.

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

[Li 2009] Z. Li, M. Gittens, S. S. Murtaza, N. H. Madhavji, A. V. Miranskyy, D. Godwin and E. Cialini: *Analysis of Pervasive Multiple-Component Defects in a Large Software System*, Proc. 25th Int'l Conference on Software Maintenance, Edmonton, Canada, September 20-26, 2009, IEEE press, pp. 265-273.

[Li 2006] J. Li, F. O. Bjørnson, R. Conradi, and V. B. Kampenes: An Empirical Study of Variations in COTS-based Software Development Processes in Norwegian IT Industry, *Journal of Empirical Software Engineering*, 11(3):433-461, 2006.

[Li 2005] J. Li, R. Conradi, O.P.N. Slyngstad, C. Bunse, U. Khan, M. Torchiano, M. Morisio: *Validation of New Theses on Off-the-Shelf Component Based Development*, Proc. 11th Int'l Symposium on Software Metrics (Metrics '05), Como, Italy, September 19-22, 2005, IEEE Press, pp. 26 - 36.

[Li 2004] J. Li, R. Conradi, P. Mohagheghi, O.A. Sæhle, Ø. Wang, E. Naalsund, and O. A. Walseth: *An Empirical Study on Component Reuse inside IT industries*, in F. Bomarius and H. Iida (Eds.): Proc. 5th Int'l Conf. on Product Focused Software Process Improvement (PROFES'2004), April 5-8, 2004, Kyoto, Japan, Springer Verlag, LNCS 3009, pp. 538-552.

[Lim 1994] W.C. Lim: Effects of Reuse on Quality, Productivity and Economics, *IEEE Software*, 11(5):23-30, September/October 1994.

[Liu 2009] D. Liu, Q. Wang, J. Xiao, *The Role of Software Process Simulation Modeling[sic.] in Software Risk Management: a Systematic Review*, Proc. Third Int'l Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, Florida, USA, October 15-16, 2009, IEEE press, pp. 302-311.

[Ma 2008] J. Ma, J. Li, W. Chen, R. Conradi, J. Ji, and C. Liu: A State-of-the-Practice Study on Communication and Coordination Between Chinese Software Suppliers and Their Global Outsourcers, *Software Process: Improvement and Practice (SPIP)*, 13(1):89-100, January/February 2008.

[Madanmohan 2004] T. R. Madanmohan and R. De: Open Source Reuse in Commercial Firms, *IEEE Software*, 21(1):62-69, January/February 2004.

[Maximilien 2003] E. M. Maximilien and L. Williams: *Assessing test-driven development at IBM*, in Proc. 25th International Conference on Software Engineering (ICSE'03), Piscataway, NJ, May 3-10 2003, IEEE CS Press, pp. 564-569.

[McIlroy 1968] M. D. McIlroy: *Mass-produced software components*, NATO Science Committee Seminal Paper, Garmisch, Germany, Springer-Verlag, 1968.

[Mens 2008] T. Mens, J. F.-Ramil, S. Degrandsart: *The Evolution of Eclipse*, Proc. 24th Int'l Conference on Software Maintenance, Beijing, China, September 28 – October 4, 2008, IEEE press, pp. 386-395.

[Microsoft 2010] Microsoft, www.microsoft.com, accessed 12 August 2010.

[Mili 1995] H. Mili, F. Mili and A. Mili: Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528-561, June 1995.

[Mockus 2000] A. Mockus and L. Votta: *Identifying Reasons for Software Changes Using Historic Databases*, Proc. Int'l Conference on Software Maintenance (ICSM '00), San Jose, California, USA, October 11-14, 2000, pp. 120 – 130.

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

[Mohagheghi & Conradi 2004a] P. Mohagheghi and R. Conradi: *An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes*, Proc. Int'l Symposium on Empirical Software Engineering (ISESE) 2004, Redondo Beach (Los Angeles), USA, August 19-20, 2004, pp.7-16.

[Mohagheghi & Conradi 2004b] P. Mohagheghi, R. Conradi, O. M. Killi, H. Schwarz, *An Empirical Study of Software Reuse vs. Defect Density and Stability*, in Proc. 26th Int'l Conference on Software Engineering (ICSE'2004), Edinburgh, Scotland, May 23-28, 2004, IEEE-CS Press, pp. 282-291.

[Mohagheghi 2006] P. Mohagheghi, R. Conradi, and J. A. Børretzen: *Revisiting the Problem of Using Problem Reports for Quality Assessment*, in Kenneth Anderson (Ed.): Proc. 4th Workshop on Software Quality, Shanghai, P. R. China, May 21, 2006 - as part of Proc. 28th International Conference on Software Engineering (ICSE) & Co-Located Workshops, May 21-26, 2006, ACM Press 2006, pp. 45-50.

[Mohagheghi & Conradi 2008] P. Mohagheghi and R. Conradi: An Empirical Investigation of Software Reuse Benefits in a Large Telecom Product, *ACM Transactions of Software Engineering Methodology (TOSEM)*, 17(3):3.1-3.31, June 2008 (extended from ICSE'04 paper).

[Morisio 2002] M. Morisio and M. Torchiano: *Definition and Classification of COTS: a Proposal*, Proc. 1st Int'l Conference on COTS-Based Software Systems (ICCBSS), Orlando, FL, USA, February 4-6, 2002, Springer Verlag, LNCS 2255, pp. 165-175.

[Niebuhr 1934] Niebuhr, 1934, printed in J. Kaplan (ed.): *Bartlett's Familiar Quotations*, 16th edition, 1992.

[O&S 2006] O&S Masterplan at Statoil ASA, http://intranet.statoil.no, 2006.

[Oberndorf 1997] T. Oberndorf: *COTS and Open Systems - An Overview*, 1997, http://www.sei.cmu.edu/str/descriptions/cots.html#ndi, accessed 18 May 2010.

[O'Connell 2006] D. O'Connell: *Boeing's Experiences using the SEI ATAM® and QAW Processes*, April 2006, http://www.sei.cmu.edu/architecture/saturn/2006/OConnell.pdf, accessed 18 May 2010.

[Odzaly 2009] E. E. Odzaly, D. Greer and P. Sage: *Software Risk Management Barriers: an Empirical Study,* Proc. 3rd Int'l Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, Florida, USA, October 15-16, 2009, IEEE press, pp. 418-421.

[OSI 1998-2010] Open Source Initiative, http://www.opensource.org/index.php, accessed 18 May 2010.

[Perry 1992] D. E. Perry and A. L. Wolf: Foundations for the study of software architecture, *ACM SIGSOFT*, 17(4):40-52, 1992.

[Paulson 2004] J. W. Paulson, G. Succi and A. Eberlein: An Empirical Study of Open-Source and Closed-Source Software Products, *IEEE Trans. Software Eng.*, 30(4):246-256, April 2004.

[Pooley 2008] R. Pooley and C. Warren: *Reuse Through Requirements Traceability*, Proc. Third Int'l Conference on Software Engineering Advances, Sliema, Malta, October 26-31, 2008, IEEE Computer Society, pp. 65-70.

[Ramesh 2004] V. Ramesh, R. L. Glass and I. Vessey: Research in Computer Science: An Empirical Study, *Journal of Systems and Software*, 70(1-2):165-176, February 2004.

[Robson 2002] C. Robson: *Real World Research: A Resource for Social Scientists and Practitioner-researchers (Regional Surveys of the World)*, 2nd edition, Blackwell Publishers, 2002.

[Ropponen 2000] J. Ropponen and K. Lyytinen: Components of Software Development Risk: How to Address Them? A Project Manager Survey, *IEEE Trans. Software Eng.*, 26(2), 98-112, February 2000.

[Ruffin 2004] M. Ruffin and C. Ebert: Using Open Source Software in Product Development: A Primer, *IEEE Software*, 21(1):82-86, Jan./Feb. 2004.

[Scacchi 2006a] W. Scacchi, J. Feller, B. Fitzgerald, S. Hissam and K. Lakhani: Understanding Free/Open Source Software Development Processes, Guest Editorial, *Software Process Improvement and Practice*, 11(2):95-105, March-April 2006.

[Scacchi 2006b] W. Scacchi: Socio-Technical Interaction Networks in Free/Open Source Software Development Processes, in S. T. Acuña and N. Juristo (Eds.): *Software*

Process Modelling, International Series in Software Engineering, volume 10, p. 1-27, Springer US, May 2006.

[Schwaber & Sutherland 2010] K. Schwaber, J. Sutherland: *SCRUM Guide*, February 2010, http://www.scrum.org, accessed 18 May 2010.

[SEVO 2004] Conradi et al.: *SEVO Project Application form 2004*, Software Evolution (SEVO) project, for the Research Council of Norway, NTNU, Trondheim, June/October 2003, http://www.idi.ntnu.no/grupper/su/epos/sevo/sevo-final-oct03.pdf, accessed 18 May 2010.

[Sjøberg 2005] D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N.-K. Liborg and A. C. Rekdal: A Survey of Controlled Experiments in Software Engineering, *IEEE Trans. Software Eng.*, 31(9):733-753, September 2005.

[Sommerville 2010] I. Sommerville: *Software Engineering*, Ninth International Edition, Pearson Addison-Wesley, 2010.

[SPICE] The Software Process Improvement and Capability dEtermination (SPICE), http://www.sqi.gu.edu.au/SPICE/, accessed 18 May 2010.

[Stallmann 2005] www.gnu.org, 2005, accessed 18 May 2010.

[Tichy 1998] W. F. Tichy: Should Computer Scientists Experiment More?, *IEEE Computer*, 31(5):32-40, May 1998.

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

[Townsend 1997] E. S. Townsend: Wells' Fargo's 'Object Express', *Distributed Object Computing*, 1(1):18-27, February 1997.

[Trac 2010] The Trac Open Source Project, http://trac.edgewall.org/, accessed 5 October 2010.

[Umholtz 1994] D. C. Umholtz and A. J. Leitgeib: Engineering Function Points and Tracking System, *STSC CrossTalk Journal*, November 1994, http://www.stsc.hill.af.mil/crosstalk/1994/11/xt94d11e.asp, accessed 18 May 2010.

[vanderBijl 2003] M. van der Bijl, A. Rensink and J. Tretmans: *Compositional testing with ioco*, Proc. 3rd Int'l Workshop on Formal Approaches to Testing of Software (FATES '03), Montreal, Canada, October 6, 2003, Springer Verlag, LNCS 2931, pp. 86-100.

[vanDeursen 2001a] A. van Deursen: *Program comprehension risk and opportunities in Extreme Programming*, Centrum Wiskunde & Informatica (CWI), Amsterdam, SEN-R0110, ISSN 1386-369X, 2001.

[vanDeursen 2001b] A. van Deursen, L. Moonen, A. van den Bergh and G. Kok: *Refactoring test code*, Proc. 2nd Int'l Conf. on Extreme Programming and Flexible Processes in Sw. Engr. (XP2001),Sardinia, Italy, May 20-23, 2001, pp. 92-95.

[vanVliet 2008] J. C. van Vliet: *Software Engineering: Principles and Practice,* 3rd Edition, Wiley & Sons, 2008.

[Vigder 1996] M. Vigder, M. Gentleman and J. Dean: *COTS Software Integration: State of the Art,* Technical Report National Research Council of Canada (NRC) No. 39190, 1996.

[Vigder 1997] M. Vidger and J. Dean: *An Architectural Approach to Building Systems from COTS Software Components,* Proc. 1997 Center for Advanced Studies Conference (CASCON 97), Toronto, Ontario, November 1997, http://seg.iit.nrc.ca/English/abstracts/NRC40221abs.html, accessed 18 May 2010.

[Voas 1998a] J. M. Voas: COTS Software – the Economical Choice?, *IEEE Software*, 15(2):16-19, March/April 1998.

[Voas 1998b] J. M. Voas: The Challenges of Using COTS Software in Component-Based Development, *IEEE Computer*, 31(6):44-45, June 1998.

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

[Weiderman 1997] Weiderman et al.: *Approaches to Legacy System Evolution*, Technical Report, CMI/SEI-97-TR-014, Software Engineering Institute, December 1997.

[Wohlin 2000] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén: *Experimentation in Software Engineering*, Kluwer Academic Publishers, 2000.

[Wong 2008] W. E. Wong, T. H. Tse, R. L. Glass, V. R. Basili and T. Y. Chen: An assessment of systems and software engineering scholars and institutions (2001-2005), *J. Systems and Software*, 81(6):1059-1062, June 2008.

[Xie 2009] G. Xie, J. Chen and I. Neamtiu: *Towards a Better Understanding of Software Evolution: An Empirical Study on Open Source Software,* Proc. 25th Int'l Conference on Software Maintenance, Edmonton, Canada, September 20-26, 2009, IEEE press, pp. 51-60.

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

[Zahran 1998] S. Zahran: *Software Process Improvement-Practical Guidelines for Business Success*, Addison-Wesley, 1998.

[Zelkowitz 1998] M. V. Zelkowitz and D. R. Wallace: Experimental models for validating technology, *IEEE Computer*, 31(5):23-31, 1998.

[Zhang 2009] H. Zhang: *An Investigation of the Relationships between Lines of Code and Defects*, Proc. 25th Int'l Conference on Software Maintenance, Edmonton, Canada, September 20-26, 2009, IEEE press, pp. 274-283.

[Aaen 2001] I. Aaen, J. Arent, L. Mathiassen and O. Ngwenyama: A Conceptual MAP of Software Process Improvement, *Scandinavian Journal of Information Systems*, 13(1):123-146, 2001.

Appendix A

In this appendix, the 6 articles that contribute the most towards the work in this thesis are presented. They are presented in the same order as already discussed earlier in this thesis. The articles are titled as follows:

- **P1:** An Empirical Study of Developers Views on Software Reuse in Statoil ASA.
- **P2:** Development with Off-The-Shelf Components: 10 Facts.
- **P3:** Preliminary results from an investigation of software evolution in industry.
- **P4:** The Impact of Test Driven Development on the Evolution of a Reusable Framework of Components – An Industrial Case Study.
- **P5:** Identifying and Understanding Architectural Risks in Software Evolution: An Empirical Study.
- **P6:** Risks and Risk Management in Software Architecture Evolution: an Industrial Survey.

The original formatting of the articles has been kept where possible.

P1: An Empirical Study of Developers Views on Software Reuse in Statoil ASA

Published in proceedings of ISESE'2006.

Odd Petter N. Slyngstad, Anita Gupta, Reidar Conradi, Parastoo Mohagheghi
Dept. of Computer and Information Science (IDI) Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
+4773593440

{oslyngst, anitaash, conradi, parastoo} at idi.ntnu.no

Harald Rønneberg, Einar Landre
Statoil KTJ/IT
Forus, Stavanger, Norway
+4751990000

{haro, einla} at statoil.com

ABSTRACT

In this article, we describe the results from our survey in the IT-department of a large Oil and Gas company in Norway (Statoil 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 focuses on components built and reused in-house. The results show that reuse benefits from the developers view include lower costs, shorter development time, higher quality of the reusable components and a standardized architecture. Component information repositories can contribute to successful software reuse. However, we found no relation between reuse and increased rework. Component understanding was generally sufficient, but documentation could be improved. A key point here is dynamic and interactive documents. Finally, quality attribute specifications were trusted for the applications using reusable components in new development, but not for the reusable components themselves.

Categories and Subject Descriptors
D.2.13 [Software Engineering]: Reusable Software- *reusable libraries.*

General Terms
Measurement, Verification.

Keywords
Empirical Study, Software reuse, CBSE.

ISESE'06, September 21-22, 2006, Rio de Janeiro, Brazil.

1. INTRODUCTION

Software reuse can be specified in two directions [14], namely *development for reuse and development with reuse*. The former refers to systematic generalization of software components for later reuse, while the latter deals with how existing components can be reused in existing and

new applications and systems. However, when it comes to reusing in-house built components, these two processes are tightly related.

Currently, we are studying the reuse process in the IT-department of a large Norwegian Oil & Gas company named Statoil ASA[1] and collecting quantitative data on reused components. To improve our understanding and collect evidence from several sources, we also performed a survey followed by semi-structured interviews in the organization. The research interests are obtained from the extant literature, and include the major benefits and factors contributing towards reuse, the effect of reuse on rework, as well as understanding and trust of component and quality specifications. Based on these issues, we have defined and explored several research questions through a survey questionnaire.

The results support some conclusions from earlier studies, while contradicting others. The sample size is rather small, and further studies will be used to refine and further investigate the research questions presented here. This study can therefore be seen as a pre-study. This paper is structured as follows: Section 2 discusses software reuse and CBSE, Section 3 has related work, and Section 4 discusses research background and motivation. Furthermore, Section 5 contains the results of our survey, Section 6 discusses these results, while Section 7 concludes.

2. SOFTWARE REUSE AND CBSE

Software reuse can have varying degrees of application, ranging from case-by-case basis (ad-hoc) to fully systematic approaches [6]. The most-inclusive definition of the term encompasses reuse of any and all assets, that is, from design and code through established procedures to documentation and knowledge. Benefits include easier understanding of the functionality, a potential shorter time-to-market, as well as possibly less effort spent on maintenance and future adoption of new requirements [17].

However, over the past decade, several attempts have been made at improving software development practices by design techniques, developing more expressive notations for capturing a system's intended functionality, and encouraging reuse of pre-developed system pieces rather than building from scratch [2]. Already in 1972, Davis Parnas wrote about the advantages of decomposing a system into modules. He mentions benefits such as [12]:

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

A new style of software development based on the principles of Parnas, and emphasizing component reuse, is CBSE; This involves the practices needed to perform component-based development in a repeatable way to build systems that have predictable properties [1]. *Component-Based Software Engineering* (CBSE) and *Component-Based Development* (CBD) are approaches to the old problem of handling the complexity of a system by decomposition, and these two concepts are often used indistinguishably [10]. Although, much effort has been devoted to define and describe

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

the terms and concepts involved, there is some literature that distinguishes between these two concepts. According to Bass [1], 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, however involves the practices necessary to perform CBD in a repeatable way to build systems that have predictable properties [1]. An important goal of CBSE is that components provide services that can be integrated into larger, complete applications.

CBSE allows the reuse of common functionality between applications, as well as organization-wide distribution of best practices. This functionality is embodied in components, which provide services that can then be included in new development, hence reused. Research has long investigated the connections between reuse and CBSE in terms of experience accumulated by practitioners on issues related to software reuse. CBSE provides the means to flexibly upgrade or replace parts of a system in order to satisfy the increasing requirements for agility and speed in new development. Another key feature of CBSE is the focus on quality attributes and corresponding testing.

3. RELATED WORK

Lim [8] have conducted a study of the effect reuse have on quality, productivity and economics in Hewlett-Packard. Data was collected from two reuse programs in this company. The results of this study revealed that reuse can provide a substantial return on investment. HP reuse programs documented improved quality, increased productivity, shortened time-to-market, and enhanced economics resulting from reuse.

Frakes & Fox [4] conducted a survey in 1991-1992, where they answered 16 commonly asked questions about reuse. A total of 113 people from 28 U.S organizations and one European organization, with a median size of 25,000 employees, participated in this survey. Some of the results that the study revealed were that education influences reuse, developers actually prefer to reuse instead of building components from scratch, reuse is more common in telecommunications compared to aerospace, and that having a reuse repository does not improve software reuse. Additionally, they found that a common software process may be advantageous.

A large reuse project was the REBOOT (Reuse Based on Object-Oriented Techniques) project [14], where the focus was on the importance of the organizational aspects of reuse in addition to the traditional technical perspective. These issues include organization and processes, as well as business drivers and human factors. The availability of more experience with industrial reuse may show the importance of these non-technical factors to be at least equal to that of the technological aspects [5] [9].

Another interesting survey is one performed by Morisio, Ezran and Tully [11]. They analyzed 24 projects in both large and small companies in Europe performed in 1994-1997 involving reuse. Their results revealed that successful component reuse was achieved when the organizations had a potential for reuse because of commonality among applications, management committed to introducing reuse process, modifying non-reuse processes, and addressing human factors.

Although the aforementioned studies cannot be directly compared to our survey, they are important to see the general trends in software reuse, and provide some of the

motivation for our research questions. We will also compare with these studies where applicable, as shown in section 6 where we discuss our results.

In [7], Li et al. investigated developer attitude towards reuse of in-house components, collecting data from 26 respondents in 3 companies. They found that the concerns were the same among those reusing components built in-house, as among those using Components-Off-The-Shelf (COTS), when it comes to (re)negotiation of requirements, documentation and the specification of quality attributes for components. Also, their results lend support to the claim that repositories do not contribute towards success in software reuse, and show that informal communication between developers can be very valuable, due to shortcomings in the component documentation. This is the work which is most applicable to our research in this survey, and a number of the questions in the questionnaire have been adapted to our use, as seen in section 4. Our contribution is hence to confirm or decline the results from the aforementioned studies, in addition to possibly reveal new results.

4. RESEARCH BACKGROUND AND MOTIVATION

Over the last decades a large push has been towards understanding the issues involved in reuse and discover the benefits and disadvantages of different approaches within the field. In CBSE, a key point of utilizing software reuse is to be able to manage software evolution through reusing components systematically, to take into account new requirements when they appear. Successful introduction and propagation of a software reuse program can be characterized by three overall points [11]:

- Commitment from management at all levels,
- Process modifications in the following manner:
  - Starting reuse processes,
  - Altering non-reuse processes,
  - Taking human factors into account, and
- Awareness of the organization's context.

When it comes to development *with* reuse, being able to match the requirements to existing reusable components is important. Also, being able to obtain sufficient knowledge of these components, as well as being able to reuse them with little or no modification, are paramount issues [9].

Our motivation is to reevaluate the issues surrounding software reuse from the perspective of developers involved in a reuse program. In particular, we want to explore the possible benefits, disadvantages and contributors towards successful reuse of software components. We also want to look at the documentation and quality specifications of reusable components that is available to the developers, who reuse them in new development. In the following section, our research questions for the survey are presented.

4.1 RESEARCH QUESTIONS

RQ1: What are the key benefits of reuse? The existing literature on software reuse claims that reuse has a positive effect on quality, productivity and time-to-market [8]. These benefits appear to be present from the second reuse occurrence; hence a positive return on investment can easily be seen over a relatively short period of time. The

purpose in our case is to confirm whether these positive effects can be seen in the same way from the perspective of the developers.

RQ2: Which factors contribute to facilitate reuse? As aforementioned, key factors towards facilitating reuse in industry are management commitment, necessary process modifications, and organization context awareness [11]. These are factors on a higher level, which developers may not have a large amount of influence on, although they affect developers directly. It may therefore be interesting and beneficial to investigate developer's opinions on this question, while still extracting extra qualitative information.

RQ3: Does reuse increase rework? Statoil ASA has it as a goal to keep rework as low as possible. Rework in the company is concerned with fixing problems (due to changes in requirements or misunderstood/ambiguous requirements), and may be more for reused components if extra effort is needed to analyze and fix such problems for components developed earlier or by other teams. It is therefore important for them to be aware of the causes and possible remedies surrounding this issue.

RQ4: Do developers have sufficient information to understand the relevant components? If the answer is no, how can they solve this problem? Component information for developers should at least encompass requirements and functional specifications. In addition, lower level details such as use cases, tests and the like can be valuable, but depending on the individual area of responsibility the needs may be different between developers. A key issue noted in literature is the need for the developers to have enough relevant information available. A noted problem within this issue is the inability to express quality attribute information on a per component basis [3].

RQ5: Do developers trust the relevant quality specification of the components? If the answer is no, how can they solve this problem? Trust is paramount in CBSE in the sense of developing trustable systems from components for which the developers may only have partial information. While CBSE allows the construction of systems from individual components, there is only a low focus on integration and quality attribute issues [18]. Here, we want to check the current status, and obtain the developers opinions about what can be done to remedy the situation, if problems exist.

The questionnaire used in this study is extended and adapted from that of an earlier study, also on reuse [7]. Some of the questions and sections have been modified and added due to different research questions.

Table 1. The questions in our questionnaire

General on software reuse[2]	**Comments**
Q6. What is your highest level of education? Q7. How many years of experience do you have with ***software development*** after completed education? Q8. How many years of experience do you have with ***reusing components*** after completed education? Q9. How important do you consider software reuse for achieving the following benefits? (5 point scale; answer alternatives: Lower development costs, Shorter development time, Higher JEF [3] component quality, A more standardized architecture, Lower maintenance costs (including technology updates), Increased knowledge/knowledge sharing, Other (please specify)) Q10. What software artifacts are most important to be reused? (Requirements, Use Cases, Design, Code, Test data/documentation ranked 1 to 5) Q11. Does Statoil ASA have a training program about software reuse? Q12. If "Yes" in Q11, have you attended this training program? Q13. Do you use a formal software reuse process for ***developing*** JEF components? Q13b. If "No" in Q13, would the availability of such a formal software reuse process be beneficial for you? Q14. Do you use a formal software reuse process for ***reusing*** JEF components? Q14b. If "No" in Q14, would the availability of such a formal software reuse process be beneficial for you?	All questions on general software reuse are customized specifically for Statoil ASA.
Requirements (re)negotiation	
Q15. Are requirements changed/(re)negotiated due the development process (DCF and S&A[4])? (5 point scale) Q16. Are requirements misunderstood / ambiguous? (5 point scale) Q16b. If "Very often" or "Often" in Q16, what are the consequences (e.g. excessive effort)? Q17. Are requirements flexible (by flexible we mean that requirements are easy to change, modify, etc.) in the development projects? (5 point scale) Q18. The requirements (re)negotiation processes related to the JEF components work efficiently in the projects (DCF and S&A)? (5 point scale) Q19. Rework (by rework we mean extra effort to fix problems due to changes in requirements or misunderstood/ambiguous requirements)	Q15, Q17, and Q18 are adapted from [7], while the remainders are customized specifically for Statoil ASA.

[2] Questions Q1-Q5 deal with the general respondent information, and the results from these questions are used towards characterizing the respondents.

[3] JEF is the name used to refer collectively to the reusable components in use at Statoil ASA.

[4] DCF and S&A are two development projects at Statoil ASA, which utilize reuse in new development.

has increased after introducing JEF components? (5 point scale)	
Value of component information repository	
Q20. Availability of a JEF repository (e.g. for storing information about JEF components) would be beneficial for me? (5 point scale) Q20b. If "Agree" or "Strongly agree" in Q20, please specify why: Q20c. If "Strongly disagree" or "Disagree" in Q20, please specify why:	Q20 is adapted from [7].
Component understanding	
Q21. Which of the following JEF components do you find the most difficult to develop and / or reuse? (The JEF components are listed, respondent is asked to rank them for both cases in terms of difficulty) Q22. How well do you know the SJEF[5] architecture? Q23. How well do you know the interface of the components? (5 point scale) Q24. How well is the design / code of the reusable components documented? (5 point scale) Q24a. If the answer of Q24 is "Poorly" or "Very poorly", is this a problem (please specify why)? Q24b. If the answer of Q24 is "Poorly" or "Very poorly", what are the problems with the documentation? Q24c. If the answer of Q24 is "Poorly" or "Very poorly", how would you prefer the documentation? Q25. What is your main source of documentation about JEF components during implementation (e.g. documentation, ask the JEF team)? Q26. Please specify if there are any other problems with understanding JEF components Q27. How do you usually reuse a JEF component? (alternatives: as is, with modifications, with modifications performed by the JEF team, Other (please specify), No relevance)	Q22, Q23, Q24, Q25, Q27 are adapted from [7], while the remainder are customized specifically for Statoil ASA.
Specification of components quality attributes (non-functional requirements)	
Q28. How are the specifications for JEF components' quality attributes defined? (5 point scale) Q29. How are the specifications for the (developed) system quality attributes defined? Q29b. If "Poorly" or "Very poorly" in Q28 and / or Q29, what can be done to improve the situation? Q30. Are the components tested for their quality attributes before integrating them with other components?	Q28, Q29, Q30 are adapted from [7].

Table 2. Research questions vs. questionnaire questions

Questions	RQ1	RQ2	RQ3	RQ4	RQ5
Q9-Q10	X				
Q6-Q8, Q11-		X			

[5] SJEF is the name given to the architecture which the reusable components are built on.

Q14, and Q20.					
Q15-Q19			X		
Q21-Q27				X	
Q28-Q30					X

Questions Q1-Q5 deal with general respondent information, and the results from these questions are used towards characterizing the respondents.

4.2 THE QUESTIONNAIRE

Based on our aforementioned research questions we have decided to use a quantitative survey, supplemented with a semi-structured interview. Our questionnaire consists mainly of check boxes, but gives also the individual respondent to contribute their own qualitative input. From the respondents own personal qualitative data we hope to obtain information, which can be supplemented with the rest of the data material. Due to our research questions, we think that a standardized survey with semi-structured interviews seems most appropriate for our data collection. This is because this research method gives us the opportunity to obtain satisfactory amount of information from each respondent with the help of a structured survey. A quantitative survey also gives us the possibility to sort out the collected data in a least time-consuming way. It gives us the opportunity to analyze data with statistical tools and analysis techniques.

Once the questionnaire was formed, it was first pre-tested on two separate occasions among 6 academic colleagues to obtain comments and to ensure that we were asking the questions understandably and would obtain the desired information. The final questionnaire is 11 pages long and contains 30 questions, which are grouped in five parts. These parts are General on software reuse, Requirements (re)negotiation, Value of component information repository, Component understanding and Specification of components quality attributes (non-functional requirements). Each question in the questionnaire has been used to study one of the research questions. Table 1 describes the questions in more detail, and the correspondence between research questions and the questions in the questionnaire is in Table 2.

4.3 THE CONTEXT

Statoil ASA is a major oil and gas operator on the Norwegian continental shelf. They are headquartered in Europe, present in 28 countries, and have 24 000 employees worldwide. Within the company, the central IT-department is responsible for developing and delivering software which is meant to give key business areas better flexibility in their operation. This department consists of approximately 100 developers worldwide, located mainly in Norway and Sweden. The 16 developers we selected are located in Norway, specifically in Stavanger, Trondheim and Oslo. Since 2004, 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, in the form of a framework based on Java Enterprise Framework components. The actual JEF framework (Java Enterprise Framework) consists of seven separate components (these are: JEF Client, JEF Workbench, JEF Util, JEF Dataaccess, JEF SessionManagement, JEF Security and JEF Integration), which can be applied separately or together when developing applications. This strategy is now being propagated to other divisions within

Statoil ASA. Reuse in Statoil ASA is component-based, with a foundation in an in-house developed architecture and with a related component framework based on Java Enterprise Framework technology.

We are currently studying two projects at Statoil ASA, namely **DCF (Digital Cargo Files)** and **S&A (Shipment & Allocation)**. The **DCF** application is mainly a document storage application. It imposes a certain structure to the documents stored in the application, and is based 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 pertaining to these information entities. DCF is meant to replace the current practice of cargo files, which are physical folders containing printouts of documents pertaining to a particular cargo or deal. A "cargo file" is a container for working documents related to a deal or cargo, within operational processes, used by all parties in the O&S strategy plan at Statoil ASA. The DCF application consists of 21459 LOC, and has a current used budget to date of 15.7 million Norwegian Kroner (about 2 million Euros). The **S&A** application aims to allow operators to carry out risk analysis on shipments from loading at terminals and offshore, as the current application is not able to take care of complex agreements (i.e. mixing of oil qualities within the same shipment). The S&A application consists of 64319 LOC, and has a current used budget to date of 17 million Norwegian Kroner (about 2.12 million Euros).

4.4 DATA COLLECTION

Data collection was carried out by two NTNU PhD students, the first and second author of this paper. We selected Statoil ASA, since they are cooperating with us in our SEVO (Software EVOlution) project and throughout our PhD research. The respondents are developers in Statoil ASA. This survey is, therefore, a non-probability sampling, based on convenience as described in Section 4.5. 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. The survey was distributed among the developers, who were then allowed a few hours within which to complete it. We had contacted and agreed upon the date with the relevant department and project managers beforehand, to ensure that enough time was allotted for this purpose. The developers answered the questionnaires separately, and they were filled out by hand. Filling out the questionnaire took 12-14 minutes, as estimated from the test runs. None of the actual respondents used more time than the allotted time to finish answering the questions. 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 for providing support with possible misunderstandings in answering the questionnaire, as well as obtaining more thoroughly qualitative information around the issues presented in the questionnaire.

4.5 RESPONDENTS

All respondents in our survey are developers that are involved in the DCF and S&A projects, and the JEF team. They all belong to the central IT-department at Statoil ASA which utilizes development *for/with* reuse in their own development projects. The software is developed mainly for other units within Statoil ASA as customers, and aims to be at the forefront of what technology can offer. 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 got 16 filled-out questionnaires back. These 16 developers were selected, since their work is related to JEF component reuse. There are a total of 100 developers in the IT-department, as aforementioned. However, these 16 developers are the only one's currently specifically involved with software reuse at Statoil ASA, and the remainders would therefore be less relevant for us in this survey.

All of the respondents have an IT background and education, seven of them have a Master of Science degree, while the other nine have education on the Bachelor degree level. A total of 22 roles were identified; 14 had a role as developer, 4 had a role as designer, 2 had a role as an architect and 1 had a role as a test manager. In addition, there was 1 respondent who filled the responsibility roles of maintenance and support. Therefore, several of the respondents had multiple roles within and also between the projects / teams. Seven of them had been working in software development between five and ten years, while the majority of the remaining respondents had more than ten years of experience. Only three respondents had less than five years overall experience. The majority expressed having less than ten years of experience in working with reuse.

5. PRESENTATION OF RESULTS

In this section, we summarize the survey results. All the statistical data presented in this study are based on valid answers, and *no relevance* answers are not included in the analysis. The statistical analysis tool we used is SPSS version 1.0 and Microsoft Excel 2003.

5.1 RQ1: What are the Key Benefits of Reuse?

First, we wanted some general information about software reuse in Statoil ASA, and questions Q6-Q14 were asked to get this information. However, Q9 and Q10 were asked to provide answer to RQ1 and are based on developers' subjective opinion related to this issue. The result of Q9 is shown in Figure 1, and the result of Q10 is shown in Figure 2. These figures are boxplots, showing the upper and lower 25% and 75 % quartiles, as well as the median, and outliers where applicable [15][16].

The numbering along the vertical axis in Figure 1 is the individual ranking; where 2=Low, 3=Medium, 4=High and 5=Very high. None of the respondents gave *very low* to the benefits. The abbreviations cq, ik, ldc, lmc, sa and sdt along the horizontal axis in Figure 1 are subsequently higher JEF component quality, increased knowledge/knowledge sharing, lower development cost, lower maintenance cost (including technology updates), a more standardized architecture and shorter development time. From Figure 1, we can see that most developers think that the component quality, lower development costs, a more standardized architecture, and shorter development time are seen as equally important benefits of software reuse, while increased knowledge/knowledge sharing is less important.

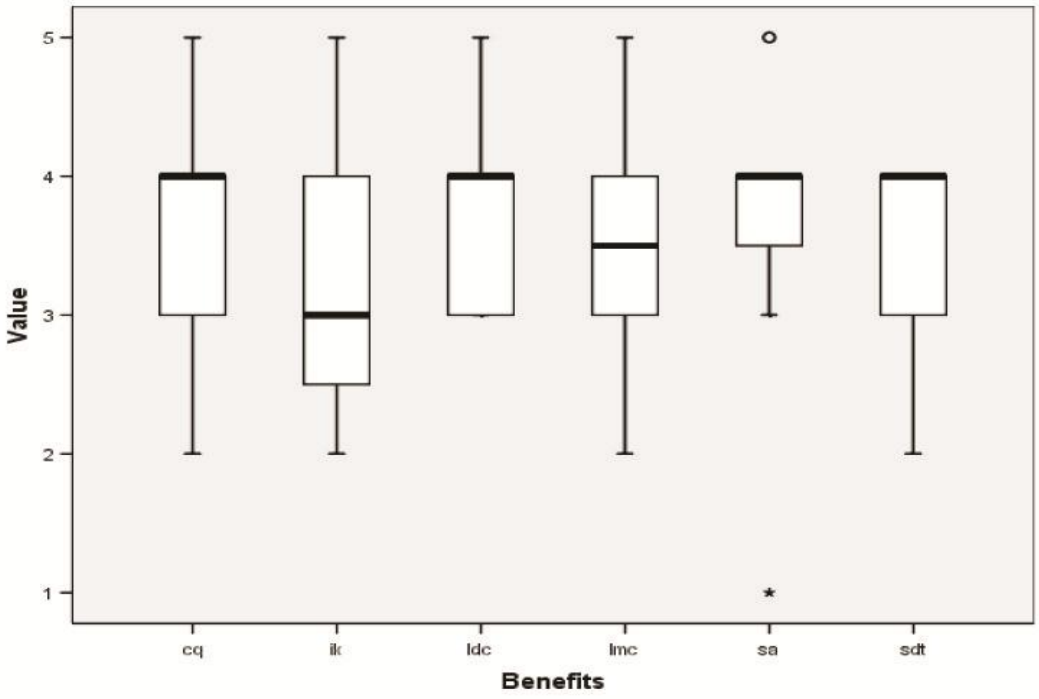

Figure 1. Benefits of software reuse

The numbering along the vertical axis in Figure 2 is the priority given by each developer; where 1=most important and 5=least important. The abbreviations co, ds, rq, te, uc along the horizontal axis in this figure are subsequently code, design, requirements, test data/documentation and use cases. From Figure 2 below, we can see that most developers think that design/code is the most important to be reused, while requirements and use cases are less important to be reused.

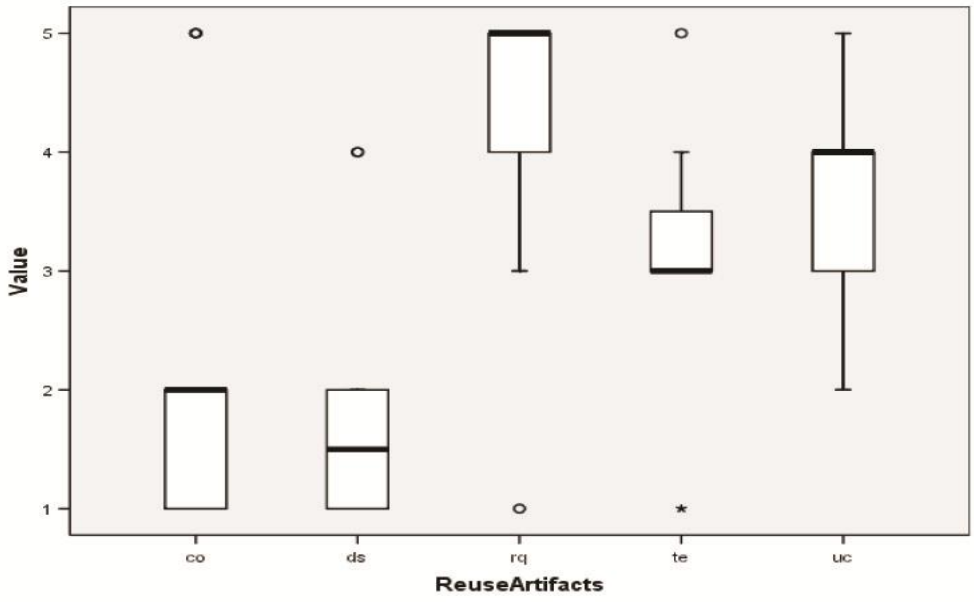

Figure 2. Software artefacts important to be reused

In summary, most of the developers think that lower development cost, shorter development time, higher JEF component quality, and a more standardized architecture are the most important benefits of reuse, while the artefacts that are important to be reused are design/code in order to achieve the benefits.

5.2 RQ2: Which Factors Contribute to Facilitate Reuse?

The questions used to evaluate this research question were Q6-Q8, Q11-Q14 and Q20. The results from Q6-Q8 (level of education and experience) are briefly presented in Section 4.5 above. From this, we can see that mean experience with software development is 8.6 years, while mean experience with software reuse is 6.9 years. Q11-Q14 revealed that 12/16 of the developers *don't know* whether Statoil ASA has a training program on software reuse. We know from communication with upper level management that Statoil ASA does not have a formal process for developing and/or reusing reusable components. Nevertheless, the questionnaire revealed some disagreement between developers on this issue, as 3/16 answered *Yes* and 5/16 answered *No* to question Q13. Likewise, in Q14, 5/16 answered *Yes* while 6/16 answered *No*.

It has already been shown thoroughly in other studies that a component repository for storing the reusable components themselves is not a contributor for reuse. We therefore decided to explore the value of a repository of information about the components. From Q20, we see that the vast majority of the respondents think that such a repository would be beneficial for them; none of them disagree, and only two said they were neutral on this issue. This may be partly due to the poor documentation of reusable components as discussed in RQ4. It should be noted that the main function of a traditional reuse repository is search and retrieval of reusable components, which is not relevant here.

The factors that contribute towards facilitating reuse can hence be summarized as follows. With education, the slight majority have a bachelor's degree, while the rest have a higher education, but we have seen no evidence that this contributes towards reuse. The same is true for experience (with software development as well as with software reuse) – here too, we have seen no indication that experience promotes reuse. Also, the majority of the developers have no knowledge of possible training programs at Statoil ASA on software reuse, and there is confusion surrounding the issue of whether formal process(es) for developing and/or reusing software is actually in use. Finally, a repository for *information about the reusable components* would be advantageous.

5.3 RQ3: Does Reuse Increase Rework?

Questions Q15-Q19 were used to investigate whether reuse leads to increased rework levels. We found that for Q15, 11/16 developers feel that the requirements are changed/(re)negotiated *somewhat* in the development projects (DCF and S&A). Further, regarding Q16, 5/16 think that the requirements are *often or very often* misunderstood/ambiguous, and the same amount think that this is *seldom* the case, while another 6/16 think is *somewhat* true. 10/16 of the developers think that requirements are *somewhat* flexible in Q17 (while the remainders said this was only *seldom* the case), and 7/16 *agree* that the process of (re)negotiation of requirements towards the reusable components works efficiently for Q18 (Here, another two

developers *disagree*, while the rest are *neutral)*. A plausible reason for this may be that requirements naturally change often in typical development projects.

Finally, when asked directly in Q19, 3/16 think that the introduction of reuse has not caused increasing rework, while 3/16 are of the opposite opinion, and the remaining are neutral. In summary, we have not found any significant evidence that reuse leads to an increase in rework, hence our results remain inconclusive.

5.4 RQ4: Do Developers Have Sufficient Information to Understand the Relevant Components? If the Answer is No, How can they Solve this Problem?

Questions Q21-Q27 were used to investigate this research question. From Q21, the most difficult components appear to be JEF Client and JEF Integration, while the easier one's are JEF Util and JEF Dataaccess.

Q22 revealed that 8/16 know the architecture *well or very well*, while 7/16 know it *somewhat*, and only 1/16 replied knowing it *poorly*. Q23 shows that 5/16 know the component interfaces *well or very well*, while 8/16 know it *somewhat*, and only 2/16 answered that they know it *poorly*.

In Q24, 2/16 think that the design/code is *well* documented, while 9/16 wrote *somewhat* and, again, 3/16 wrote *poorly*. In Q25, developers answered that their main sources of information on the JEF components are typically *the JEF team, javadoc/source code, colleagues and the JEF homepage*. Furthermore, in Q26, when asked about other problems with component understanding, the answers were that *the reusable components are immature/unstable as they are changing, and that documentation is insufficient, as well as that it is unclear how different components cooperate and the dependencies between them*. Finally, Q27 shows that 8/16 of the developers reuse the JEF components *"as-is"*, while 3/16 *reuse with modifications by the JEF Team*. Only 2/16 replied that they *"reuse with modifications"* (performed by themselves).

The most difficult components are JEF Client and JEF Integration, and about half of the developers have a good understanding of the architecture. However, 50% of the developers only know the component interfaces somewhat, and 56% think that the design/code is somewhat well documented. Another 50% reuse the JEF components *"as-is"*.

It hence appears that while the majority of the developers have knowledge of the component architecture as well as the component interfaces, they're still unhappy with the documentation that is available. Currently, the qualitative answers from the questionnaire and the semi-structured interviews reveal that developers would like a website with overview, tutorial, sample code and good javadoc of the component code, which is as interactive as possible for the developers.

5.5 RQ5: Do Developers Trust the Relevant Quality Specification of the Component? If the Answer is No, How can they Solve this Problem?

In this research question, we used Q28-Q30 to elicit the answers. Q28 reveals that while 8/16 developers think that the quality attributes for the JEF components are *poorly or very poorly* defined, 3/16 think that they are *well or very well* defined. However, Q29 shows that when it comes to the projects DCF and S&A, 12/16 think that the respective quality attributes are *well or very well* defined, while only 2/16 think

they're *poorly or very poorly* defined. Lastly, in Q30, 6/16 developers *do not know* whether the components are tested for fulfilment of their quality attributes before integrating them with other components, while 5/16 think that this is only *sometimes* done. 4/16 also think that this is *always* done, while 1/16 think that this is *not done at all*.

In summary, 75% of the developers say that the quality attributes (non-functional requirements) for the development projects (DCF and S&A) are well defined, while 50% think that these quality attributes are not well defined for the reusable JEF components. Hence, the developers trust the quality specifications for the development projects, but not for the reusable JEF components. In order to remedy this problem, the qualitative answers from the questionnaire and the semi-structured interviews show that the specification and publication of quality attributes for the reusable components should be improved in terms of realism and clarity. Additionally, more consistent component testing was also suggested as a way to handle this problem.

6. DISCUSSION OF THE RESULTS

We now discuss our research questions based on the results from our survey, as well as the inherent limitations and validity threats.

6.1 RQ1: What are the Key Benefits of Reuse?

Lim [8] showed that key benefits of reuse can be seen in terms of higher quality, higher productivity, and shorter time-to-market as well as economic benefits. Our results confirm that in the view of the developers, lower development costs (economic benefit), shorter development time (productivity – hence shorter time-to-market), higher JEF component quality (quality), are perceived as the key benefits of reuse. Additionally, a standardized architecture is also seen as a benefit.

6.2 RQ2: Which Factors Contribute to Facilitate Reuse?

Frakes & Fox [4] asked about whether reuse education both in academia and in industry, influences reuse. They found that though reuse education in academia and industry helps towards reuse, it is still uncommon in academia, as well as in industry. Our results show that many of the developers do not know about the existence of a reuse training program, so Statoil ASA must become better at promoting such training programs where they exist.

They [4] also wrote that although their respondents say that a common software process does not promote reuse, it may nevertheless contribute indirectly. Our results show that though Statoil ASA does not have a formal process specifically for developing/reusing reusable components, they do have one for general software development, which can implicitly affect reuse positively. Here too they must also become better at informing their developers about this.

When it comes to a repository, literature has concluded that a reuse repository does not increase levels of code reuse [4] [11]. We investigated the question of whether the availability of an JEF repository (e.g. for storing information regarding JEF components, rather than the components themselves) would be beneficial. On this issue, our results show an overwhelming agreement from the developers. The qualitative reasons given include easier information sharing, easier learning, improved

level of documentation, better overview of the documentation and functionality, as well as of typical existing problems and troubleshooting.

We would like to emphasize again here that we have investigated the issue of documentation through an *information* repository, not for "finding" reusable components.

6.3 RQ3: Does Reuse Increase Rework?

The theoretical foundation behind this research question is that because extra effort may be needed towards analyzing and fixing problems with reusable components, reuse could potentially increase rework (as discussed in section 4.1). The analysis here is inconclusive, as we cannot show any link between reuse and increased rework. Possible reasons for this are as follows. The development projects DCF and S&A are meant to reuse the JEF components developed by the JEF team, however, our results show that developers often have multiple responsibility roles that often cross project and team lines. This means that there is no clear division between development *for/with* reuse in Statoil ASA. Reused components are developed internally and the organizational flexibility improves knowledge and compensates for the lack of a specific reuse process.

As aforementioned in Section 5.3, we have not seen any indications that reuse leads to an increase in rework, and our results here are therefore inconclusive.

6.4 RQ4: Do Developers Have Sufficient Information to Understand the Relevant Components? If the Answer is No, How can they Solve this Problem?

Li [7] found that developers understood the components well, despite a lack of related documentation, and that the knowledge was instead gotten through prior experience and local experts. Our results support these findings, in that the majority of the developers have sufficient understanding about the relevant components. They too think that the documentation could be better (see section 5.4), and use the JEF team or previous experience to achieve the necessary component understanding.

6.5 RQ5: Do Developers Trust the Relevant Quality Specification of the Component? If the Answer is No, How can they Solve this Problem?

Here, literature reports that most developers are unhappy with the quality specification of the components [7], and therefore cannot use this information. Our results, however, show that the relevant quality specification for the development projects DCF and S&A are well-defined, while that for the JEF components are not. This could be caused by more rapidly changing requirements, resources, personnel involved in the JEF team, and poor documentation. It may also be difficult to define quality specifications on the component level.

6.6 Threats to Validity

We here discuss the possible threats to validity in our survey, using the definitions provided by Wohlin [19]:

Construct Validity: Most of the research questions and the actual questions in the questionnaire have their origin from the research literature. From these, 12 of the survey questions were adapted towards our survey. Further, through pre-testing among local colleagues, most of the questions were refined additionally. Also, terms that may be unfamiliar to the respondents have been defined in the questionnaire handout.

External Validity: Another threat is that our survey is done completely by convenience sampling. That is, we chose this group of developers specifically because they are working with software reuse in the two projects DCF and S&A, as well as the JEF Team, which we are already involved in studying. It should be noted that the company's IT-department has a total of 100 developers, and that we have only sought answers from 16 of these. Nevertheless, these 16 are all the developers that are currently involved with software reuse at Statoil ASA. Also, the applications (DCF and S&A), with JEF development included, are representative of typical applications developed in-house at Statoil ASA in terms of size and allocated resources. Nevertheless, our limited sample size should be kept in mind. This means at least that we cannot generalize outside the context.

Internal Validity: The respondents were asked to answer the questionnaire by their project leader, and a contact relationship with them as well as with upper level management already existed at the time the questionnaire was carried out. The company itself has an expressed interest in gaining knowledge from the answers to the survey. We therefore are of the opinion that the respondents have answered truthfully to the best of their ability. In addition, we also provided support for possible ambiguities of the questions in the questionnaire.

Conclusion Validity: This analysis is performed based on an initial collection of data. Though too small a sample to be statistically significant, it still yields interesting and valuable insights for us and for Statoil ASA.

7. CONCLUSION AND FUTURE WORK

We have investigated the opinions of developers on software reuse, related to the five main areas: *benefits of reuse, factors contributing towards reuse, possible relations between reuse and increased rework, component understanding and quality attribute specification.* Overall, our results can be summarized as follows:

- When it comes to the *benefits of reuse*, the results of RQ1 show that the benefits of reuse can be seen in terms of lower costs, shorter development time, higher quality of the JEF components and a standardized architecture. These results support those found in literature [8].
- In terms of *factors contributing towards reuse* (RQ2), we found no link to education. We also found no evidence that experience contributes towards reuse. When it comes to formal processes, our findings support the literature [4] in that though the formal process in use is only for software development in general (not specifically for software reuse), this may still have an implicit positive effect. The results also show improving documentation of the reusable components would have been largely beneficial towards achieving successful reuse.
- On RQ3, we found no relation between *reuse and increased rework*, hence we cannot come to a conclusion. This is possibly caused by the mandate of reuse in the company, along with the multiple responsibility roles that often cross project and team lines, and that there hence is no clear division between development *for* reuse and development *with* reuse in the company.
- The results of RQ4 showed most developers have sufficient *understanding of the components*, but the documentation could be improved, as they largely use the JEF team or prior experience to achieve the required component understanding. A key point here is dynamic and interactive documents.

- *Quality attribute specification* (RQ5) was shown to be trusted for the development projects developing with reuse, but insufficient for the reusable components. This may be caused, in our case, by the rapid changes in the team that develop the reusable components, in terms of requirements, resources and personnel.

Our investigation is dependent on the subjective opinions of the developers as respondents. The results are presented to Statoil ASA and contribute to improving their processes. One interesting question raised from their side is whether the results of this work can be used as input to future larger reuse programs. The results will be combined with other research in the company to explain findings regarding reuse. We also plan to expand our dataset with more respondents, to refine the research questions based on our initial findings, and to compare our survey results with the actual results of the company.

8. ACKNOWLEDGMENTS

This work has been done as a part of the SEVO project (Software EVOlution in component-based software engineering), and ongoing Norwegian R&D project from 2004-2008 [13], and as a part of the first and second authors' PhD study. We would like to thank Statoil ASA for the opportunity to collect data from their reuse projects. We also thank our local colleagues for feedback and all of the respondents.

9. REFERENCES

[1] Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K. *Volume I: Market Assessment of Component-based Software Engineering* in SEI Technical Report number CMU/SEI-2001-TN-007, 2001, (http://www.sei.cmu.edu/)

[2] Brown, A. W., Wallnau, K.C. The Current State of CBSE. *IEEE Software, 15*, 5 (Sept/Oct 1998), 37-46.

[3] Crnkovic, I. Component-based Software Engineering – New Challenges in Software Development. In *Proc. 25th Int'l Conference on Information Technology Interfaces* (Cavtat, Croatia, June 16-19, 2003). IEEE Press, 2003, 9-18.

[4] Frakes, W. B. and Fox, C. J. 16 Questions on Software Reuse. *CACM, 38*, 6 (June 1995), 75-87.

[5] Kim, Y., Stohr, E. A. Software Reuse: Survey and Research Directions. *Journal of Management Information Systems, 14*, 4 (Spring 1998), 113-147.

[6] Kruger, C. Software Reuse. *ACM Computing Surveys, 24*, 2 (June 1992), 131-138.

[7] 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.): Proc. 5th Int'l Conf. on Product Focused Software Process Improvement (PROFES'2004)* (Kansai Science City, Japan, April 5-8, 2004). Springer Verlag LNCS 3009, 2004, 538-552.

[8] Lim, W. C. Effect of Reuse on Quality, Productivity and Economics. *IEEE Software, 11*, 5 (Sept./Oct. 1994), 23-30.

[9] Mili, H., Mili, F., Mili, A. Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering, 21*, 6 (June 1995), 528-561.

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

[11] Morisio, M., Ezran, M., Tully, C. Success and Failure Factors in Software Reuse. *IEEE Transactions on Software Engineering, 28*, 4 (April 2002), 340-357.

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

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

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

[15] Stevens, S. *Psycholoical Scaling: Theory and Applications.* Wiley, 1951.

[16] Tukey, J. W. *The Collected Works of John W. tukey, Vol. III: Philosoply and Principles of Data Analysis: 1949-1964.* Wadsworth & Brooks/Cole Advanced Books & Software, 1986.

[17] Vliet, H. V. *Software Enginering, Principles and practice,* Wiley, 2nd edition, 2001.

[18] Councill, B., Heineman, G. T. Component-Based Software Engineering and the Issue of Trust. In *Proceedings of the 22nd International Conference on Software Engineering* (Limeric, Ireland, June 4-11, 2000). ACM Press, 2000, 21-31.

[19] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., Wesslén, A. *Experimentation in Software Engineering – An Introduction.* Kluwer Academic Publishers, 2002.

P2: Development with Off-The-Shelf Components: 10 Facts

Published in IEEE Software, March/April 2009

Jingyue Li, Reidar Conradi, Christian Bunse, Marco Torchiano, Odd Petter N. Slyngstad, Maurizio Morisio

jingyue@idi.ntnu.no, conradi@idi.ntnu.no, Christian.Bunse@i-u.de, marco.torchiano@polito.it, oslyngst@idi.ntnu.no, maurizio.morisio@polito.it

Abstract

The paper summarizes the results of several industrial surveys on issues related to the development of systems using Commercial-Off-The-Shelf and Open Source Software components. The results demonstrate the following. (1) There is a discrepancy between academic theory and industrial practices regarding the use of components. One reason is that researchers have empirically evaluated only a few theoretical methods; hence, industrial practitioners currently have no reason to adopt them. Another reason might be that researchers have specified the contexts of application of only a small number of theories in sufficient detail to avoid misleading users. (2) Academic researchers often hold false assumptions about industry. For example, research on requirement negotiations often assumes that a client will be interested in, and be capable of, discussing the technical details of a project. However, in practice this is usually not true. In addition, the quality of a component in the final system is often attributed solely to component quality before integration, ignoring quality improvements by integrators during component integration.

Keywords: COTS-based development, OSS-based development, empirical studies.

0. INTRODUCTION

A software component (henceforth, component) is a unit of code that integrators can combine with other components and integrate into a system in a predictable way. Software developers build components on the principle of "build once, reuse often". Hence, the use of components promises to reduce development time and cost while increasing software quality. An IDC survey in early 2007 illustrates that more than 50% software developers have used software components for development in the most recent projects [1].

Components from third parties (so called Off-The-Shelf (OTS) components) are of different types, i.e. Commercial-Off-The-Shelf (COTS) and Open Source Software (OSS), which makes composition a complicated task that requires risk-management techniques. In principle, OTS component-based development involves three stakeholders: component provider (i.e. COTS vendor or OSS community), application integrator, and application client. Different stakeholders face different issues and challenges. Application integrators must manage processes and knowledge well to ensure successful component selection, component integration, and component maintenance. To meet such goals, integrators need to communicate with component providers to get information and support. They also need to coordinate with clients to determine requirements as well as to get the OTS-based system accepted. Figure 1 summarizes software development with OTS components from the integrators' perspective.

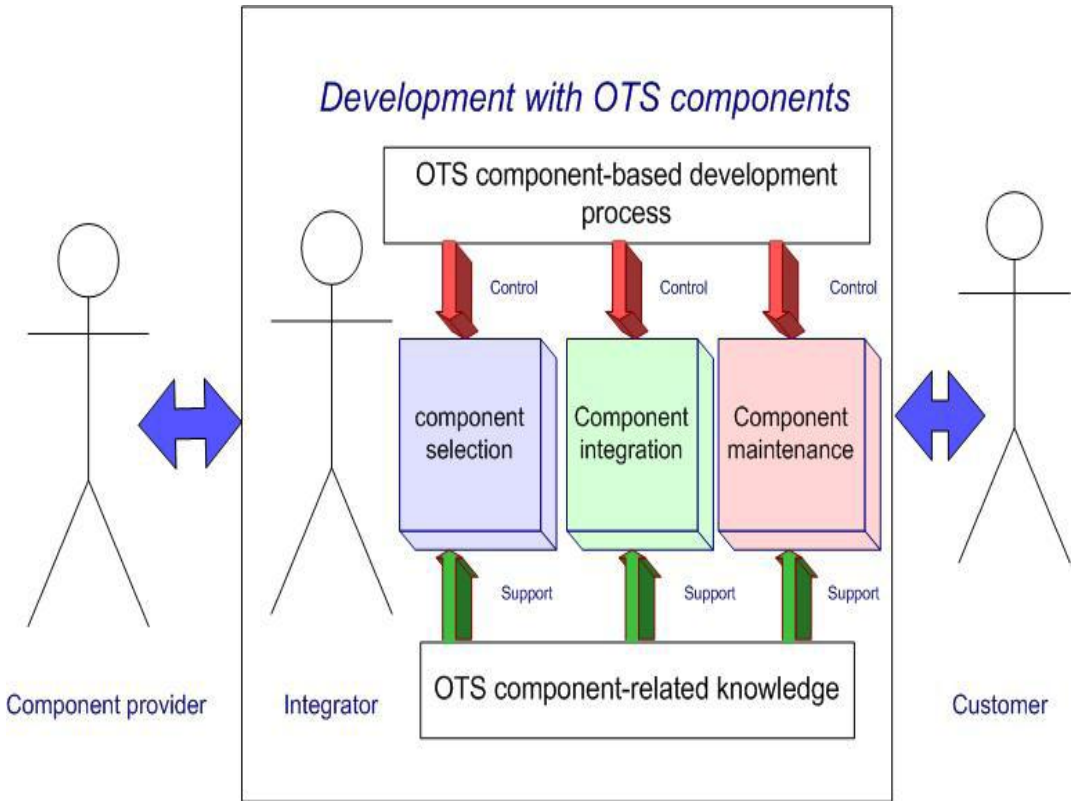

Figure 1. Development with OTS components: actors and activities

Researchers have proposed several methods for improving processes and managing risks to facilitate the integration of OTS components [2] [3]. However, they have evaluated few of these by industrial case studies [4]. *This makes it difficult for project managers to evaluate the effectiveness of proposed methods and to make the right decisions on the basis of empirical evidence.*

With these problems in mind, we performed a series of studies (see the side bar) to investigate the state of the practice in OTS-based development and the reasons for applying these practices. We here report the results of the last two steps of our studies: (i) an industrial survey with 133 completed projects from 127 companies, and (ii) 28 follow-up telephone interviews. Detailed information of the participated companies and projects is in [5].

1. Ten facts about industrial practices on OTS component-based development

Our findings illustrate that there are discrepancies between the proposals of academic researchers and industrial practice.

1.1 FACT 1

Development process: *Companies use traditional processes enriched with OTS-specific activities to integrate OTS components.*

Boehm et al. regard neither the waterfall model nor the evolutionary development model as suitable for COTS-based development [2]. When using the waterfall model, integrators identify requirements at an early stage and choose COTS components at a later stage. This increases the likelihood that COTS components will not offer certain features that are required. Evolutionary development assumes that additional features can be added quickly if required. However,

developers find it difficult to upgrade COTS components: the vendor will not change the product upon the request of a single client, and the absence of source code prevents the development team from adjusting or adapting the COTS components. This would suggest that companies need to adapt their development processes in response to using OTS components.

In fact, out of 75% (100/133) of the projects we investigated in the main study, developers chose their main development processes before they even started to think about using OTS-components.

Why do not companies adapt the development process? Four out of the 28 companies interviewed in the follow-up study are immature and have no well-documented development processes. They insisted on their ad-hoc development processes without considering any changes due to the use of OTS components. They regarded the introduction of formal and heavyweight development techniques proposed in the literature as ***useless***, because OTS components either have been integrated several times in previous projects, or do not constitute important parts of the overall system. Surprisingly, five interviewees in the follow-up study thought that there was no difference between selecting/integrating OTS components and selecting/integrating software classes from an internal library. Two participants using agile processes believed that adaptation was not necessary, because agile means lightweight and depends solely on developers' experience.

Side bar: Genesis of the study

Our studies were inspired by a qualitative study [6] of COTS usage in seven IT companies in Norway and Italy, done in 2002. The study identified six "theses" on COTS usage, partly challenging commonly held beliefs. To build upon this study, we first conducted a qualitative **prestudy** *[7] in 2003, using a structured interview. We interviewed project managers of 16 projects from 13 Norwegian IT companies to summarize their lessons learned from COTS-based development. To verify the conclusions of our prestudy, we then developed a quantitative* **main study** *[5], which we performed as a survey in 2004 and 2005 to address IT companies in Norway, Italy and Germany. In this survey, we investigated the process improvement and risk management issues in 133 (47 from Norway, 48 from Germany, and 38 from Italy) completed COTS or OSS component-based projects that we selected using a stratified-random sampling strategy [8]. The response rate of this study shows that 53% of investigated software development companies had already completed OTS component-based projects by 2005. Results of this survey illustrate the state of the practice of OTS component-based development. To determine the reasons for phenomena discovered in the main study, we conducted a* **follow-up** *study, using telephone interviews with 28 participants (six in Norway, 12 in Germany, and 10 in Italy) selected by convenience.*

If companies adapt processes, how do they do so? They typically added a prototyping phase during OTS selection to evaluate and learn about OTS components. One of them uses the RCPEP process proposed in [9]. Two others are moving toward the new version of the V-model, i.e. V-model XT, which has been explicitly adapted for use with OTS components.

Our insights: Familiarity with the OTS component is proposed as a leading factor to be considered when selecting OTS component [6]. Our results show that the familiarity with OTS candidates is also an important factor to be evaluated for customizing the whole development process. Although companies are using adapted evolutionary (e.g. RCPEP) and waterfall (e.g. V-model XT) processes to integrate OTS components, sufficient knowledge with OTS candidates may make the usage of these adapted processes unnecessary.

1.2 FACT 2

> ***Component selection:*** *Integrators select OTS components informally. They rarely use formal selection procedures.*

Researchers have proposed several procedures for formally selecting COTS components that promise 'fail-safe' decisions (see, for example, [3] for a summary). However, when we analysed the data from our main study, we found that, in practice, integrators habitually select components in an ad-hoc manner, using in-house expertise and/or web-based search engines.

Why do not companies use formal selection processes? We found that in some cases, integrators had simply neglected steps for searching and evaluating OTS components. The interviewees gave two reasons: (i) only a limited number of OTS candidates were available in the market, and (ii) the integrators' company already had a long-term partnership with a specific provider. However, about 20% of interviewees in the follow-up study were unaware of the formal selection processes proposed by academia. Most interviewees were sceptical about the cost-effectiveness of using a formal process, especially under time-to-market pressure. They would rather trust the experience of in-house expertise than any formal process, as discovered in [6]. In agile projects, integrators did not consider formal OTS selection processes at all, because they believed that adopting any kind of formal procedure would undermine the agile nature of the development process.

If a formal selection process was applied, what was done? Only one of the 28 interviewees stated that his company had used a formal process for selecting OTS components. The project on which this interviewee worked was safety- and performance-critical. A candidate component had to fulfil several quality requirements and had to follow strict industrial standards. Integrators adopted candidates on the basis of either client recommendation or a search of other sources. They then narrowed down the number of candidates by reading the literature and scanning discussion boards. After that, they purchased and installed the remaining candidates and used them in a small prototype project, in order to evaluate both non-functional properties and functionalities against requirements. Integrators also evaluated the components' compliance with the given industrial standard.

Our insights: Researchers have evaluated few formal processes for selecting OTS components empirically, with the aims of measuring their cost-effectiveness and determining their applicability in certain contexts. Thus, the pre-conditions and benefits of using a formal process are unclear. Without evidence on the possible benefits, integrators are reluctant to use formal processes, which are supposed to be complex and time-consuming.

1.3 FACT 3

> ***Component selection:*** *There is no specific phase of the development process in which integrators select OTS components. Selecting components in early phases has both benefits and challenges.*

Researchers usually suggest that integrators select components in an early phase of development, so that they can identify possible problems early. The data from our main study showed that most integrators did select OTS components in the early phases of a project, e.g. prestudy (38%), requirement specification (30%), and overall design (16%). However, some integrators selected OTS components in later phases, i.e. detailed design (6%), and even coding (7%). (Three percent of participants did not know when the component was selected).

Reasons for, and issues pertaining to, selecting OTS components in the prestudy phase: One reason is that the component will drive the definition of the architecture of the whole system, as discovered in [6]. Several interviewees had bad experiences of system architecture restructuring when they had selected components in later phases. Furthermore, integrators often decide to (re)use familiar components in the prestudy phase. However, some interviewees recognized that if they select a component from a set of unknown OTS components in such an

early phase, they must use comprehensive documentation and the results of trials. Yet documentation is often absent and, when it exists, often does not describe the actual component accurately. This, in turn, requires unexpected extra effort in later phases. Moreover, at the beginning of a project, integrators do not know all the required functions of a system. Thus, they may have to write adapters or change OTS components later.

Reasons for, and issues pertaining to, selecting OTS components in the requirements or design phase: Most integrators selected OTS components during the requirements or overall design phase. The interviewees identified benefits of selecting OTS components in these phases as follows:

- The integrators know the system architecture and the functional requirements for a possible component.
- They thus have a solid basis on selecting OTS components.
- Integrators can readily adapt the system to the specific needs of the component, thus enabling integration to proceed seamlessly.
- Once the integrators have defined the architecture and have selected the OTS candidates, they can easily define test-cases and make systematic plans for quality assurance.
- Once the integrators have defined the requirements and have selected the OTS candidates, they can estimate the cost of the project more accurately.

However, selecting OTS components in these phases carries certain recognized risks and challenges, as follows:

- Integrators usually do not care enough about technical details in an early stage. This may subsequently lead to problems of implementation and integration.
- During the course of a long project, the providers of an OTS component may release a new version during the detailed design or coding phases. Consequently, the integrators may need to re-evaluate the component and redesign the system.
- In projects using agile development processes, integrators typically identify and document requirements by means of "user stories", and set up the entire process in such a way that they can make changes easily. Therefore, the earliest possible phases at which integrators should think about components are either the detailed architectural design or the development iterations. However, this may require integrators to expend extra effort on refactoring the system.

Our insights: Most approaches assume that selecting components in the early phases of a project will yield benefits [10]. However, we have identified pitfalls that integrators must consider if they wish to select OTS components in the early phases of a project.

1.4 FACT 4

> ***Component integration:*** *Estimators use personal experience when they estimate the effort required to integrate components and most of the time they do not estimate accurately. Stakeholder-related factors will affect dramatically the accuracy of estimates.*

In 83% of the 133 projects that we examined in the main study, the estimations of the effort required for integration were unsatisfactorily estimated. Only four out of the 28 interviewees used a formal effort estimation tool, e.g. COCOTS [11]. The remaining interviewees estimated the integration effort solely on the basis of personal experience.

Reasons for inaccurate effort estimation. In addition to the usual factors that may affect the accuracy of effort estimation, the following factors also contributed to inaccurate effort estimations of projects examined in the follow-up study:

- It takes time to understand how to use the components correctly, because technical details of OTS components are not described explicitly in their documentation.
- The clients changed their requirements significantly. It is difficult to satisfy changed requirements due to the inflexibility of OTS components.
- The OTS provider did not respond quickly to required changes. Integrators waste a lot of time waiting for the providers to respond.
- The OTS provider released new versions of OTS components during the project. The integrators then had to expend extra effort on adapting the system to the new versions or on evaluating and integrating them.

Our insight: Some estimation tools, e.g. COCOTS [11], take into account both the technical nature of the components and a number of issues mentioned above, e.g. component understandability and vendor response time. Our data show that estimation tools should also take into account possible changes in requirements and the evolution of components, especially for large projects with long durations.

1.5 FACT 5

Quality of the integrated system: *Negative effects of OTS components on the quality of the overall system are rare.*

The quality of OTS components is expected to be at least as good as that of in-house built components [12]. Our data support these expectations. The traditional qualities (reliability, performance, and security) of OTS components were a problem in the final system for few of the projects that we investigated.

Reasons for positive feedback on the quality of OTS components. Some interviewees in the follow-up study stated that their system was of high quality because the integrators evaluated and tested the OTS components carefully in the selection phase. Others stated that their experience with specific OTS components and the strategy of only using mature OTS components were helpful. However, another very important reason for the integrator's positive feedback on the quality of OTS components is that their expectations are not very high, either because the OTS components play only a minor role in the composed system, or because the integrators accept "minor problems" with free or low-cost components.

Our insight: In traditional software development, the quality of software is measured by how well it satisfies the client's requirements. For OTS components, there are two clients: a direct client (i.e. the application client) and an indirect client (i.e. the application integrator). When measuring the quality of components, people still refer to how well the component satisfies application client requirements after it has been integrated [12]. Our findings illustrate that, for various reasons, for example, low cost, application integrators sometimes accept OTS components that are of less than perfect quality. It is the integrator's quality assurance effort during selection and integration that ensures the quality of the OTS component in the final system.

1.6 FACT 6

OSS and COTS components: *Integrators usually used OSS components in the same way as commercial components, i.e. without modification.*

People often assume that the commercial vendors of COTS components sell a copyright license with agreed specific support and do not make the source code available, while the open source communities that provide OSS components offer freely accessible source code yet promise no specific support. The study [6] illustrates that COTS component debates should include open source component. Results of our study supported observations of [6] and found that one third of the companies we investigated in the main study do have access to the source

code of COTS components. However, only 15% of the COTS component integrators and 36% of the OSS component integrators changed the source code. In most cases, they used the OTS component "as-is".

Reasons for changing the source code in the projects investigated in the follow-up study. Integrators have to wait too long for providers to update their components.

Reasons for not changing the source code in the projects investigated in the follow-up study:

- Source code is unavailable.
- The fast and effective OTS component support renders change unnecessary.
- Developers lack the deep knowledge and thorough documentation that is required to change the source code.
- It is difficult to get changes accepted into OTS component in later releases. Integrators do not want to change the source code, so that they can "drop-in" replacements when the next version of component is released. In-house changes run the risk that the system will be incompatible with later component updates and that maintenance will become too difficult.
- Changing source code may generate legal issues. For example, the integrators have to take responsibility for any problems caused by the changed components.

Our insights: Changing the source code of OSS components may not be feasible, especially for a long-term commercial system with a possibly long evolution path ahead. Thus, application contexts, e.g. commercial vs. non-commercial application and long-term vs. short-term application need to be considered when deciding to use OSS or COTS components.

1.7 FACT 7

> ***Locating defects is difficult:*** *Although problems with OTS components are rare, the cost of locating (i.e. within or outside OTS components) and debugging defects in OTS-based systems is substantial.*

Although integrators are, in general, satisfied with the quality of OTS components (see fact 5), in 80% of the projects that we investigated integrators experienced difficulty in locating defects when they occurred.

Reasons for inefficient defect location: The following factors can cause failures within an OTS-based system: defects in OTS components, misuse, or defects in the code to integrate OTS components with other parts of the system (besides defects in the subsystems built in-house). If the documentation is incomplete or imprecise, or the source code is inaccessible, unfamiliar, insufficiently commented, or messy, application integrators find it difficult to locate the defects by themselves. Asking the provider for help may create new problems. Component providers are usually reluctant to read the code from application integrators to locate defects; especially when components from different providers are mixed. One interviewee tried test-driven development to mitigate this problem. He was unsuccessful because it is difficult to write test cases for an OSS component without in-depth knowledge of its code.

Our insight: The variety of deployment environment and configuration of OTS components hinders OTS providers to reproduce the reported errors. The irreproducible errors usually will not be prioritized and fixed by OTS providers. To improve the debugging efficiency of OTS-based systems, integrators need to work collectively with OTS providers by engaging in a process of constructing a context where the OTS provider can reproduce the reported error [13]. Moreover, integrators need to investigate relevant descriptions with the reported error from mailing list, web forums, and bulletin board in order to provide more information to and convince OTS provider that the error may potentially affect many systems.

1.8 FACT 8

Relationship with the provider: *The relationship with the OTS component provider involves much more than defect fixing during the maintenance phase.*

Researchers have claimed that a good relationship with the provider is essential for a successful OTS component-based project [14]. However, most previous studies emphasize only the technical support from providers in the maintenance phase, e.g. fixing defects. Data from our study show that additional issues related to providers need to be considered.

Issues related to component providers. In the *component selection* phase, integrators need to both evaluate component candidates and to evaluate and consult with their providers. This evaluation includes not only the reputation of a provider, but also such matters as technical support, market share, and company size. Several interviewees stated that, in their experience, OSS communities that have large user groups usually provide better support than those with smaller ones. In addition, the integrator and provider must consider legal and licensing issues and specify them in the contract. Typical example are the response times of COTS vendors (e.g. on problem reports) and the responsibility for returning code changes of OSS components. In the *component integration* phase, OTS providers need to be contacted to ease debugging, to provide extra functionalities, and to fix defects. Integrators believe that knowing a specific person at the COTS vendor company or in the OSS community is essential for reducing integration costs. The readability, accuracy, and completeness of the component documentation made available by the provider also affect the efficiency of integration. In the *maintenance phase*, integrators need the provider's help for debugging defects and for suggestions/hints on component evolution or on reusing the component in the future.

Our insights: Different people in OSS communities may be involved in different tasks to support the use of a component. For example, some senior OSS community members, who have better views on the similarities and differences between their components and others, may help OSS users to make the right evaluation and selection. Other community members, who have solid experience of fixing bugs and customizing software features, may help to ease integration. It is therefore important for integrators to know the right persons for a specific task in an OSS project, to share detailed experiences with them regularly, and to build partnership with them [4]. Hence, OSS projects need to specify contact persons on the basis of possible user needs.

1.9 FACT 9

Relationship with the client: *Involving clients in OTS component decisions is rare and sometimes infeasible.*

OTS components seldom satisfy all of a client's requirements; hence, researchers regard the (re)negotiation of requirements with the client as an important strategy in OTS-based development. However, our data from the main study show that integrators rarely involve clients in the "build vs. acquire" decision, or in selecting OTS components.

Reasons for not involving clients: Half of the companies that we investigated in the follow-up study are software houses, which produce software for the general market. Thus, they have no direct client with whom to confer when developing their products. The others develop software for a dedicated client, but most of their clients have either no interest in discussing, or insufficient technical capabilities to discuss, such issues. Only two out of 28 interviewees had involved their clients in discussing the outcome of selecting OTS components. In one project, the client was mainly interested issues pertaining to reselling and licensing, cost, or compliance with given industry standards. In the other project, the client had to be involved because the project included cooperative and distributed development. The client had to centralize OTS component selection to ensure that all the subcontractors could understand and use the selected OTS components.

Our insight: Application clients usually only care about the final products and typically are not interested in the technical details of the implementation. Most approaches to the (re)negotiation of requirements that researchers have proposed simply assume that application clients have enough background competence to discuss technical details [14]. We encourage companies to clarify, at the start of the project, the clients' interests and technical capabilities so that they can decide on possible strategies for (re)negotiation.

1.10 FACT 10

Knowledge management: *Knowledge that goes beyond the functional features of OTS components must be managed.*

The success of OTS-based development projects requires that companies manage the implicit and explicit knowledge about OTS components. More than half of the companies that we investigated in the main study already have dedicated staff (so-called "component uncles") to keep the OTS component-related knowledge. Most of them are experienced software architects and senior developers.

Which knowledge needs to be kept and shared? (1) Companies need to capture knowledge about a component itself, e.g. basic functionality, standards conformance, side-effects, undocumented issues, and non-functional properties. (2) Companies need to manage knowledge about how to facilitate component integration: licensing and reselling obligations, examples of the code that is used to connect the OTS component to the system (gluecode), and descriptions of possibilities for optimization. (3) Companies need to acquire and store information about the stakeholders. This will include client preferences, with whom to negotiate at the client side, whom to contact at the provider side, and who knows which components at the integrators' organization.

Which knowledge management mechanisms to choose? The software market changes quickly and OTS components have short release cycles. Hence, the knowledge that developers need to acquire and share will change quickly as well. In order to accommodate this changing demand, some of the interviewees advocated storing and sharing tacit knowledge through personal communications, e.g. coffee-meetings, internal seminars, informal discussion forums, or regular meetings in (agile) development groups. Other interviewees preferred more formal and recordable approaches to mitigate the problem of losing experience when a key person leaves. Some of them even claim that every project should have a touchdown meeting where they can share their collective experience. Several of the companies that we investigated in our follow-up study have set up a small Wiki site to share knowledge. Companies have also used a central authority (i.e. an OTS team or "component uncle") to manage OTS-related knowledge and yellow pages to record "who knows what". Integrators regard these as effective mechanisms for managing knowledge.

Our insight: Our results show that implicit and explicit knowledge about OTS components has been partly managed within the organization by "component uncles". However, there are very few centralized external channels for OTS users to share and communicate experience between organizations. External experiences of using certain OTS components are scattered in several COTS or OSS portals, bulletin boards, or mailing lists. Searches in search engines usually yield huge, unwieldy sets of results. A centralized experience portal for sharing OTS component-related knowledge between organizations, probably using a global OTS Wiki [15] could be a solution.

2. Conclusion

The results of our industrial surveys have revealed gaps between theory and practice regarding the use of OTS components. We suggest that researchers need to be more precise about the assumptions and contexts of the application of their proposals regarding the revision of OTS-based development processes, the processes by which companies should select their components, and the processes by which integrators and providers negotiate requirements. We further suggest that researchers conduct more empirical case studies, to investigate cost-effectiveness of proposed theories. We suggest integrator to collaborate more actively with OTS providers to facilitate debugging the defect. We also suggest integrators to investigate the strategies for (re)negotiating requirements with clients at the early stage of the OTS-based project.

Our surveys also reveal several issues that researchers need to address. By what means can providers and integrators share knowledge of OTS components on a global scale? How can people working on the field establish the "who to contact" yellow pages for each OSS project, to facilitate support from OSS communities?

As an important caveat, note that we have, thus far, collected only a small amount of data. We were the first to perform such an empirical study using a random sample of IT companies. Researchers need to perform further studies, both to validate our results and to align them with the latest progress in the field.

3. References

[1] http://www.idc.com/getdoc.jsp?containerId=IDC_P644, 2007.

[2] B. Boehm et al., "COTS integration: Plug and Pray?" *Computer*, vol. 32, no. 1, 1999, pp. 135-138.

[3] K. RPH. Leung et al., "On the Efficiency of Domain-Based COTS Product Selection Method," *J. Information and Software Technology*, vol. 44, no. 12, 2002, pp.703-715.

[4] J. Norris, "Mission-Critical Development with Open Source Software: Lessons Learned," *Software*, vol. 21, no. 1, 2004, pp. 2-9.

[5] J. Li et al., "Validation of New Theses on OTS-Based Development," *Proc. 11th Int'l Symp. Software Metrics*, IEEE, 2005, pp. 26.

[6] M. Torchiano et al., "Overlooked Facts on COTS-based Development," *Software*, vol. 21, no. 2, 2004, pp. 88-93.

[7] J. Li et al., "An Empirical Study of Variations in COTS-based Software Development Processes in Norwegian IT Industry," *J. Empirical Software Engineering*, vol. 11, no. 3, 2006, pp. 433-461.

[8] R. Conradi et al., "Reflections on conducting an international CBSE survey in ICT industry," *Proc. 4th Int'l Symp. Empirical Software Engineering*, IEEE, 2005, pp. 214-223.

[9] P. K. Lawlis et al., "A Formal Process for Evaluating COTS Software Products," *Computer*, vol. 34, no. 5, 2001, pp. 58-63.

[10] I. Crnkovic et al., "Component-based development process and component lifecycle," *Proc. 27th Int'l Conf. Information Technology Interface*, IEEE, 2005, pp. 591-596.

[11] C. Abts et al., "COCOTS: A COTS Software Integration Cost Model - Model Overview and Preliminary Data Findings," *Proc. 11th ESCOM Conf.*, 2000, pp. 325- 333.

[12] J. M. Voas, "Certifying Off-the-shelf Software Components," *Computer*, vol. 31, no. 6, 1998, pp. 53-59.

[13] Ø. Thomas et al., "Debugging Integrated Systems, an Ethnographic Study of Debugging Practice," *Proc. 23rd Int'l Conf. Software Maintenance,* IEEE Press, 2007.

[14] L. C. Rose, "Risk Management of COTS Based Systems Development," Springer LNCS, vol. 2693, 2003, pp. 353-373.

[15] C. Ayala et al., "Open Source Collaboration for Fostering Off-The-Shelf Components Selection," Proc. 3rd Int'l Conf. Open Source Systems, Springer, 2007, pp. 17-30.

P3: Preliminary results from an investigation of software evolution in industry

Published in the ERCIM workshop proceedings 2006.

Odd Petter N. Slyngstad, Anita Gupta, Reidar Conradi, Parastoo Mohagheghi,
Thea C. Steen, Mari T. Haug
Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
{oslyngst, anitaash, conradi, parastoo} at idi.ntnu.no

Harald Rønneberg, Einar Landre
Statoil KTJ/IT
Forus, Stavanger
{haro, einla} at statoil.com

Abstract

In the SEVO (Software EVOlution) project, we explore the field of software evolution in terms of software quality attributes, their characteristics and possible relations between them. Currently, we have explored preliminary data from a software engineering program in a Norwegian company (Statoil ASA), on the frequency of defects and changes of reused components. These measures are the stated quality foci of the program, and our results indicate that while defect-density evolves decreasingly over time, change-density does not exhibit a conclusive behavior. This is part of on-going research, and the results will be expanded and verified in later following publications. Overall, we aim to use the collected data towards discovering and explaining characteristics related to software evolution.

Keywords: *CBSE, software evolution, quality attributes*

1. Introduction

The purpose of the SEVO (Software EVOlution) project [SEVO, 2004] is to explore software evolution in Component-Based Software Engineering (CBSE) through empirical research. Aiming to increase our knowledge and understanding of underlying issues and challenges in software evolution, one long-term purpose of the project is to provide possible solutions to these problems. Another goal is to help industrial software engineers to improve their efficiency and cost-effectiveness in developing software based on reusable components, as well as in their ability to develop and use reusable assets. Underlying all this is the need for evidence to support or reject existing and proposed hypotheses, models, design decisions, and the like. Such evidence is best obtained through performing empirical studies in the field, and experience from such studies will be possible to incorporate into a knowledge base for use by the community.

Currently, we are studying the reuse process in the IT-department of a large Norwegian Oil & Gas company named Statoil ASA[6] and collecting quantitative data on reused components. The research questions are obtained from the existing literature, and include how the defect-density in reusable components evolves over time, as well as how the number of changes per reusable component evolves over time. Based on these issues, we have defined and explored several research questions and hypotheses through an empirical study. Here, we perform a preliminary analysis of data on defect-density and change-density of reusable components from a software engineering program in Statoil ASA, a major international petroleum company. We have chosen these two attributes for measuring software quality as they are part of the stated quality focus for the program in Statoil ASA. The purpose of this study is to gain initial understanding of software evolution from the viewpoint of these quality attributes.

The number of change requests and trouble reports is to some extent small, and future studies will be used to refine and further investigate the research questions and hypotheses presented here. This study is therefore a pre-study. This paper is structured as follows: Section 2 introduces terminology, Section 3 discusses our contribution to Statoil ASA, as well as the research context at the company. Furthermore, Section 4 introduces our research questions and preliminary data analysis, and Section 5 summarizes and discusses these preliminary results. Section 6 contains planning for further data collection and future work, while Section 7 concludes.

2. Terminology

CBSE is a new style of software development, emphasizing component reuse, which involves the practices needed to perform component-based development in a repeatable way to build systems that have predictable properties [Bass et al., 2001]. An important goal of CBSE is that components provide services that can be integrated into larger, complete applications.

Software evolution can be defined as: "….the *dynamic behaviour* of programming systems as they are *maintained* and *enhanced* over their *life times*…." [Kemerer & Slaughter, 1999]. The first studies found in literature on software evolution, were undertaken by Lehman on an OS360 system at IBM [Lehman et al., 1985]. Software evolution is closely related to software reuse, since reuse is often employed to achieve the aforementioned positive effects when evolving a system. It should be noted that several alternative uses of the term software evolution exist; some use the term to encompass both the initial development of the system and its subsequent maintenance, while others use it exclusively about the events after initial implementation, in concurrence with its original focus [Kemerer & Slaughter, 1999]. Lastly, there is some work on software evolution taxonomy [Verhoef, 2004], the author sees software maintenance as subpart of software evolution.

Software maintenance is the updating incurred on already existing software in order to keep the system running and up to date. During their lifetime software systems usually

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

need to be changed to reflect changing business, user and customer needs [Lehman, 1974]. Other changes occurring in a software system's environment may emerge from undiscovered errors during system validation, requiring repair or when new hardware is introduced.

Software maintenance can hence be:

- *corrective* (correcting faults),
- *preventive* (to improve future maintainability),
- *adaptive* (to accommodate alterations related to platform or environment), *or*
- *perfective* in response to requirements changes or additions, as well as enhancing the performance of a system

[Sommerville, 2001] [Pressman, 2000].

In summary, some see the perfective and adaptive parts of software maintenance as part of *software evolution* [Sommerville 2001]. That is, that it encompasses both aspects of modified and added scope, as well as environmental adaptations. This does not include platform changes, which are commonly referred to as porting, instead of software evolution [Frakes & Fox, 1995]. There is, hence, no clear agreement on the definition of software evolution. Although there seems to be more agreement on the definition of the different types of software maintenance, a clear distinction between software maintenance and software evolution remains elusive.

Statoil ASA [Statoil ASA O&S Masterplan, 2006] has chosen to use defect-density and change-density (stability) as indicators of software quality. A lowered defect-density shows an increased quality, while stability in terms of change-density means a stable level of resources are needed towards adaptation and perfection of the software. In this study, it is these two measures (defect-density and change-density) we will be focusing on, in order to show how the reusable components evolve over time.

3. Our contribution to Statoil ASA, and The context

Our direct contribution is helping Statoil ASA central software development unit in Norway with defining metrics, collecting data and analyzing it. We will also be contributing towards reaching a better understanding and management of software evolution, by exploring whether that employment of reusable components can lead to better system quality[7]. Finally, we expect that our results will be possible to use as a baseline for comparison in future studies on software evolution.

Statoil ASA is a large, multinational company, in the oil & gas industry. It is represented in 28 countries, has a total of about 24,000 employees, and is headquartered in Europe. The central IT-department in the company is responsible for developing and delivering software, which is meant to give key business areas better flexibility in their operation. They are also responsible for operation and support of IT-systems at Statoil ASA. This department consists of approximately 100 developers worldwide, 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

[7] The quality focus at Statoil ASA is defect density and change-density (stability).

systematically, in the form of a framework based on JEF (Java Enterprise Framework) components. This IT strategy was started as a response to the changing business and market trends, and in order to provide a consistent and resilient technical platform for development and integration [14]. The strategy is now being propagated to other divisions within Statoil ASA. The JEF framework itself consists of seven different components. Table 1 gives an overview of the three JEF releases, and the size of each component in the three releases.

Table 1: Size of JEF components, in #LOC

Component	Release 2.9	Release 3.0	Release 3.1
JEF Client	7871	8400	8885
JEF Dataaccess	181	181	268
JEF Integration	958	958	958
JEF Security	1588	1593	2374
JEF Util	1312	1359	1647
JEF Workbench	4187	4515	4748

Release 2.9 spanned the time between 09.11.2004 - 14.06.2005, while Release 3.0 spanned the time between 15.06.2005 - 09.09.2005 and Release 3.1 spanned the time between 10.09.2005 - 18.11.2005.

These JEF components can either be applied separately or together when developing applications. In total, we will be studying the architectural framework components, as well as two projects which use this framework. Here, we present a pre-analysis, reporting on preliminary results of studying defect-density and stability (change density) of 6 of the 7 reusable architectural framework components, over three releases. These three releases exist concurrently, and the data is mainly from system/integration tests. The limited dataset used in this preliminary analysis is due to current data availability.

4. Research questions and Preliminary data analysis

All the statistical data presented in this study are based on valid data, as none were missing data. The statistical analysis tools we used were SPSS version 14.0 and Microsoft Excel 2003. Our preliminary research questions are regarding defect-density and stability, and are formulated as follows:

RQ1: How does the defect-density in reusable components evolve over time?
Defect-density (number of Trouble Reports/KLOC) may be seen as belonging to the corrective maintenance category by some researchers, but maintenance can also be seen as part of evolution [Verhoef, 2004]. Therefore, measuring defect-density may help characterize the evolution of the different JEF components over time. The following are the related hypotheses for RQ1:

- *H10: The defect-density in JEF components do not change with time.*
- *H1A: There is a difference in defect-density for JEF components over time.*

RQ2: How does the number of changes per reusable component (stability) evolve over time?

Research has demonstrated that reusable components are more stable (has a lower change-density) and that this does improve with time [Mohagheghi et al., 2004]. We have chosen to use change-density (number of Change Requests/ KLOC) as an indication of the stability, as this is the defined quality focus of Statoil ASA. The following are the related hypotheses for RQ2:

- *H20: The change-density in JEF components does not change with time.*
- *H2A: There is a difference in change-density for JEF components over time.*

4.1. RQ1: How does the defect-density in reusable components evolve over time?

For RQ1 we want to see how the defect-density in JEF components evolves over time, so we decided to use ANOVA test, as this is suitable for comparing the mean defect-density between the three releases. With this test, we wanted to investigate whether there is a difference in defect-density for JEF components. To investigate this research question, all submitted defects for each component were counted, per release. We then calculated the defect-density, as the number of trouble reports (TR's) divided by kilo lines of code (KLOC) for each component. Table 2 shows the results of this calculation for three releases, all involving major changes to the software components.

Table 2: Defect-density per JEF component, in #TR/KLOC

Component	Release 2.9	Release 3.0	Release 3.1
JEF Client	17.1516	1.5476	0.1190
JEF Dataaccess	11.0497	0.0000	0.0000
JEF Integration	3.1315	0.0000	0.0000
JEF Security	5.6675	0.6277	0.0000
JEF Util	1.5244	0.0000	0.0000
JEF Workbench	3.8214	0.8859	0.2106

Here, we want to test if there is a significant difference in the mean-values of the different releases, which we are using as groups in the analysis. Table 3 shows that the average defect-density decreases with time. The significance level is 0.05, and the data was checked for normality.

Table 3: Average defect-density per release

Groups	Mean
Release 2.9	7.058
Release 3.0	0.510
Release 3.1	0.055

The ANOVA test we performed yielded a F_0 value of 7.749, and the critical value was computed to be $F_{0.005,\ 2,\ 15}$= 3.682, with a P-value of 0.0049. Since 7.749 > 3.682, it is

possible to reject the null hypothesis. In summary, we can reject H10 in favour of our alternative hypothesis H1A, and hence support the notion that the defect-density decreases with time.

The data trend for RQ1 reveals a declining defect-density, possibly caused by a corresponding decrease in change-density. We will, however, be expanding and verifying our hypothesis on defect-density with more empirical data in future work.

4.2. RQ2: How does the number of changes per reusable component (stability) evolve over time?

For RQ2 we want to see how the number of changes per JEF component evolves over time, so we again decided to use ANOVA test, as this is suitable to compare the mean change-density between the three releases. With this test, we wanted to investigate whether there is a difference in change-density for JEF components. To investigate this research question, all change requests were sorted according to JEF component and then counted, per release. We then calculated the change-density, as the number of change requests (CR) divided by kilo lines of code (KLOC) for each component. Change requests in this context mean new or changed requirements. Table 4 shows the results of this calculation.

Table 4: Change-density per JEF component in #CR/KLOC

Component	Release 2.9	Release 3.0	Release 3.1
JEF Client	13.4672	0.8333	0.2251
JEF Dataaccess	0.0000	0.0000	11.1940
JEF Integration	3.1315	1.0438	0.0000
JEF Security	9.4458	1.8832	0.6072
JEF Util	4.5732	0.7358	0.0000
JEF Workbench	8.3592	1.1074	0.0000

Here too, we decided to use an ANOVA test, to see if there was a significant difference in the mean-values of the different releases. Table 5 shows the variation in mean change-density over time. The significance level is 0.055, and the data were checked for normality.

Table 5: Average change-density per release

Groups	Mean
Release 2.9	6.496
Release 3.0	0.934
Release 3.1	2.004

As seen from Table 5, Release 3.0 has a lower change density than Release 3.1, indicating that the change-density may not simply decrease with time. The ANOVA test we performed yielded gave a F_0 value of 3.540, and the critical value was computed to be $F_{0.055, 2, 15}$= 3.682, with a P-value of 0.055. Since 3.540 < 3.682, it is not possible to

reject the null hypothesis. In summary, we cannot reject H20 in favour of our alternative hypothesis H2A.

Nevertheless, upon inspection of the data from Table 4, we see that for all components the change-density is lower in the following release, except for JEFdataaccess. In fact, the value JEFdataaccess has in Release 3.1 differs considerably compared to the other results. This may have specific explanation(s), which will be explored in later analysis.

5. Summary and Discussion of preliminary results

In Table 6, we have summarized our analysis results, along with corresponding research questions and hypotheses.

Table 6: Summary of the results

Research Questions	Hypotheses	Results
RQ1	*H10: The defect-density in JEF components do not change with time.*	H10: Rejected
	H1A: There is a difference in defect-density for JEF components over time.	H1A: Not rejected
RQ2	*H20: The change-density in JEF components does not change with time.*	H20: Not rejected
	H2A: There is a difference in change-density for JEF components over time.	H2A: Not rejected

On change-density, the data indicate that a decrease over time for five of the six components investigated. However, we are unable to conclude without further empirical data and analysis. When it comes to defect-density, our results indicate a distinct difference over subsequent releases of the JEF components. The data trend here is towards a sharp decrease. Additional trends in the size data vs. the data on change-density and defect-density exist (e.g. that some of the components have zero change density while their code size still shows an increase, or that some have high change-density while still zero defect-density) will be investigated further with more empirical data in future work. An additional possible relationship to be explored is whether large increases in change-density affect defect-density negatively, though such an effect is not indicated in the data from our preliminary analysis.

Lower defect-density means less correction are needed, and thereby a higher quality level is achieved for the reusable JEF components. When it comes to change-density, stability is important to achieve stable evolution and hence allowing for stable resources being assigned to adapt and perfect the reusable JEF components. In this way, these quality attributes can be used to partially model evolution, as they show how the quality of the reusable JEF components evolves over time.

5.1. Threats to validity

We here discuss the possible threats to validity in our study, using the definitions provided by [Wohlin, 2002]:

Construct Validity: The metrics we have used (defect-density and change-density) are thoroughly described and used in literature. Nevertheless, our definition and use of the term change-density is different from that in other studies. All our data are of pre-delivery change requests and trouble reports from the development phases for the three releases of the reusable components.

External Validity: The object of study is a framework consisting of only seven components, and the data has been collected for 3 releases of these components. Our results should be relevant and valid for other releases of these components, as well as for similar contexts in other organizations.

Internal Validity: All of the change requests and trouble reports for the JEF components have been extracted from Statoil ASA by us. Incorrect or missing data details may exist, but these are not related to our analysis of defect-density and change-density. We have performed the analysis jointly with the Microsoft Excel and SPSS tools.

Conclusion Validity: This analysis is performed based on an initial collection of data. This data set of change requests and trouble reports should nevertheless be sufficient to draw relevant and valid conclusions.

6. Planning for further data collection and Future work

So far, Statoil ASA has collected data on Trouble Reports (TR's) and Change Requests (CR's) for the reusable JEF components over several releases. They are also going to collect data on TR's and CR's for systems developed with the JEF components – so far two systems are reusing JEF components in development. Further releases of JEF components will also follow, and data will be collected on these.

In this article we have seen that there are differences in defect-density and change-density over subsequent releases, without further analysis on the differences or their causes. In further work we will be exploring these issues in more detail, as well as the possible cause-effect relation between defect-density and change-density, as well as the relation to other quality attributes. The focus may also change to encompass towards reuse and maintenance, in addition to evolution.

7. Conclusions

We have performed a preliminary investigation of how the quality attributes defect-density and change-density evolves over time for reusable components. While prior research has shown reusable components to be more stable (having a lower code modification rate) across releases [Mohagheghi et al., 2004], change density as defined in this context has not been studied before.

The overall results from our study are:

- For **RQ1, "How does the defect-density in reusable components evolve over time?",** our results show a clear difference over releases of the JEF components. The data trend here shows a sharp decline.

- On **RQ2, "How does the number of changes per reusable component evolve over time?",** our investigation on change-density shows that we cannot conclude without further data and analysis. However, the general trends in the data indicate that the change-density does decrease with time for five of the six components investigated.

In particular, lower defect-density results in less corrections being needed, hence yielding a higher level of quality of the reusable components. Such a reduction is expected if components undergo few changes between releases, e.g. in our dataset, some components have zero change-density between releases. A stable change-density is a factor towards allowing a stable evolution, hence resources used in adapting and perfecting the reusable components can be better allocated. Hence, we see that evolution can be partially modelled by looking at defect-density and change-density, as they show how the quality of the reusable components evolves over time. Our results cannot currently support these thoughts to the full extent, as the results of studying defect-density shows a decline, but the results from studying the change-density so far cannot be used to conclude, despite the apparent trends in the data. We presume that more empirical data will remedy this problem.

The SEVO project is ongoing research and this paper is meant to present preliminary results, while results will come later. We ultimately aim to look at how to reach higher quality, by demonstrating that understanding and managing software evolution can lead to better system quality.

8. Acknowledgement

This work has been done as a part of the SEVO project (Software EVOlution in component-based software engineering), an ongoing Norwegian R&D project from 2004-2008 [SEVO project, 2004-2008], and as a part of the first and second authors' PhD study. We would like to thank Statoil ASA for the opportunity to be involved in their reuse projects.

9. References

[SEVO, 2004], *The SEVO project*, NTNU, Trondheim, 2004 – 2008, http://www.idi.ntnu.no/grupper/su/sevo/.

[Bass et al., 2001] L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau, *Volume I: Market assessment of Component-based Software Engineering*. SEI Technical Report number CMU/SEI-2001-TN-007 (http://www.sei.cmu.edu/)

[Lim, 1994] W. C. Lim, *Effect of Reuse on Quality, Productivity and Economics* in IEEE Software, 11(5):23-30, Sept./Oct. 1994.

[Mohagheghi et al., 2004] P. Mohagheghi, R. Conradi, O. M. Killi, H. Schwarz, *An Empirical Study of Software Reuse vs. Defect Density and Stability* in Proc. 26th Int'l Conference on Software Engineering (ICSE'2004), 23-28 May 2004, Edinburgh, Scotland, pp. 282-291, IEEE-CS Press Order Number P2163.

[Morisio, Ezran, Tully, 2002] M. Morisio, M. Ezran, C. Tully, *Success and Failure Factors in Software Reuse* in IEEE Transaction on Software Engineering, 28(4):340-357, April 2002.

[Frakes & Fox, 1995] W. B. Frakes and C. J. Fox, *Sixteen Questions About Software Reuse*, CACM, 38(6):75-87, June 1995.

[Kemerer & Slaughter, 1999] C. F. Kemerer and S. Slaughter, *An empirical approach to studying software evolution*, IEEE Transactions on Software Engineering, vol. 25, issue 4, 1999.

[Lehman et al., 1997] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, W. M. Turski, *Metrics and Laws of Software Evolution – The Nineties View*, Proc. Fourth Int. IEEE Symp. on Software Metrics, Metrics 97, Albuquerque, New Mexico, 5-7 Nov. 1997, pp 20-32.

[Postema, Miller, Dick, 2001] M. Postema, J. Miller and M. Dick, *Including Practical Software Evolution in Software Engineering Education*, 14th IEEE Conference on Software Engineering Education and Training (CSEET'01), 19-21 February, 2001, Charlotte, North Carolina, USA, pp. 127-135.

[Sommerville, 2001] I. Sommerville, *Software Engineering*, Sixth Edition, Addison-Wesley, 2001.

[Statoil ASA O&S Masterplan, 2006] O&S Masterplan at Statoil ASA, http://intranet.statoil.no

[Pressman, 2000] R. S. Pressman: *Software Engineering: A Practitioner's Approach,* fifth edition, 2000, McGraw-Hill.

[Verhoef, 2004] Chris Verhoef, Software Evolution: A Taxonomy, http://www.swebok.org/stoneman/version_0.1/KA_Description_for_Software_Evolution_and_Maintenance(version_0_1).pdf

P4: The Impact of Test Driven Development on the Evolution of a Reusable Framework of Components – An Industrial Case Study

Published in the proceedings of ICSEA'2008.

Odd Petter N. Slyngstad[1], Jingyue Li[1], Reidar Conradi[1],
1) Department of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU), Trondheim, Norway
{oslyngst, jingyue, conradi} at idi.ntnu.no

Harald Rønneberg[2], Einar Landre[2], Harald Wesenberg[2],
2) Statoil KTJ/IT, Forus, Stavanger / Rotvoll, Trondheim
{haro, einla, hwes} at statoilhydro.com

Abstract

Test Driven Development (TDD) is a software engineering technique to promote fast feedback, task-oriented development, improved quality assurance and more comprehensible low-level software design. Benefits have been shown for non-reusable software development in terms of improved quality (e.g. lower defect density). We have carried out an empirical study of a framework of reusable components, to see whether these benefits can be shown for reusable components. The framework is used in building new applications and provides services to these applications during runtime. The three first versions of this framework were developed using traditional test-last development, while for the two latest versions TDD was used. Our results show benefits in terms of reduced mean defect density (35.86%), when using TDD, over two releases. Mean change density was 76.19% lower for TDD than for test-last development. Finally, the change distribution for the TDD approach was 33.3% perfective, 5.6% adaptive and 61.1% preventive.

1. Introduction

In this study, we are connecting Test Driven Development (TDD) together with corrective (i.e. defects) and non-corrective software changes. That is, this study is an investigation of the change characteristics of this approach for reusable components. There are but a few case studies on the effect of TDD on defects (i.e. corrective changes) in an industrial setting [3]. Also, the effect of TDD on changes (here meaning non-corrective changes) appears not to have been explicitly investigated earlier. TDD is expected to promote software reuse, since a higher focus on testing and refactoring of the code is inherent to the practice of TDD. Improved quality (i.e. lower defect density), but lower productivity has been shown in other industrial investigations [3] for TDD over test-last development, for non-reusable components. Furthermore, earlier investigations have not explicitly investigated reusable components, which may have higher requirements towards predictability, stability and maintainability [14]. Our goal in this study is to see whether these benefits can be shown for reusable components. More specifically, we investigate the effects of TDD in terms of number and type of defects, changes and the relation between these two in a framework comprised of reusable components.

Our object of study is a framework of reusable components in place in the IT-department of a large Norwegian Oil & Gas company (StatoilHydro ASA[8]). In 2003 the company defined a reuse strategy towards new development, and has since followed this strategy successfully. This strategy entails employing a framework of reusable components. The latest development in connection with this reuse strategy is the application of TDD as an integral part of their development methodology. The framework is called JEF, and has been used in the development of several new systems since its inception. Towards the end of 2005 it became apparent that a new focus was needed in order to facilitate improvement of the framework's architecture and make it more reusable, both in terms of the services provided and the components used. In response to this need, TDD was employed in new development on the framework, for versions 4 and 5. We measure defect density in terms of Trouble Reports per non-commented Source Lines of Code, and change density in terms of Change Requests per non-commented Source Lines of Code.

Earlier industrial studies on TDD for non-reusable components have shown benefits e.g. in terms of lower defect density. These studies indicate a higher software quality over traditional development [3], but also up to 16 % decrease in productivity (i.e the TDD approach required more effort). Our results show that the mean defect density was reduced by 35.86%, and mean change density to be reduced by 76.19%, for TDD in comparison to test-last development. Also, on the relationship between defect density and change density, the relationship appears near-linear related for both development methods.

This paper is structured as follows: Section 2 discusses background and related work, Section 3 has research design. Furthermore, Section 4 contains the results, Section 5 the discussion and Section 6 concludes.

2. Background and Related Work

Test Driven Development is a practice related to agile software development which entails composing unit testing prior to the actual implementation of the code [2]. All test cases must successfully pass prior to considering the implementation of new code to be complete. Also, new test cases are always added to encompass recent defects found prior to correcting such defects. The focus is on writing formalized tests for the smallest increments of functionality, then implementing that functionality, and repeating the process until the system is built [1]. TDD tests generally focus on low-level unit-like testing, rather than cross-cutting or combining testing concerns. Also, the same person fills the roles of test writer and software developer.

Several advantages of using TDD can be outlined [2]:

- Code comprehension: TDD aids in comprehending the code since, developers explain their code through test cases and code itself instead of more formal documentation.
- Efficiency: TDD makes it easier to determine the problem source when encountering new defects (i.e. during development).

[8] ASA stands for "allmennaksjeselskap", meaning Incorporated. Statoil merged with Hydro creating StatoilHydro in 2007.

- Testing: The test cases developed using TDD comprises important assets towards further testing as well as the identification of newly found defects (as noted above).
- Reduces defect injection: According to Hamlet and Maybee [12], software maintenance and debugging in traditional test-last development is commonly considered a low-cost activity where the code is patched, but the design and the specifications are neither examined nor changed accordingly. Such small code changes [12] can be up to 40 times more likely to cause further errors, and new faults are commonly injected during debugging and maintenance. As TDD encourages the inclusion of new test cases to counter newly found defects, the amount of defects caused by e.g. code maintenance can be reduced.

There are also some disadvantages seen in using TDD:

- Design: TDD commonly includes no or little design. This works well only for well-written and –understood code, and enables the possibility of lacking conceptual integrity. This means that when defects are found, there is no "backup" in terms of formal design and documentation [7], and one may miss the "big" picture [9][10] and thereby incur problems related to the architecture.
- Context: The amount of effort used in writing test cases is considerable, and may be context-dependent [2].
- Refactoring: Refactoring is used extensively to manage complexity when utilizing TDD [2].
- Level of skill required: A high level of experience and knowledge is needed in order to develop and maintain the test assets in TDD [7] [8].

George and Williams [2] carried out a set of controlled experiments using 24 professional pair programmers. Using a small Java program as test object, they found that the code developed using TDD allowed passing of 18% more functional black-box test cases. The authors claim this shows the code is of higher quality when using TDD. On the other hand, using TDD required about 16% more time than for the control group using a water-fall like approach.

Maximilien et al [Maximilien 2003] performed a case study at IBM, where TDD was put in use as development methodology. They report reduced defect densitys of about 50% compared to ad-hoc testing, as well as minimal impact on development productivity as the project completed on time. Furthermore, the automated test case suite that was created during the project functioned as a reusable and extendable asset towards future quality improvements. Another case study at IBM using TDD was reported by Williams et al. [6], where TDD was employed to reduce defects. They found that there were 40% fewer defects, and that the team's productivity was not affected by the additional test-first focus. Furthermore, they also comment that TDD provides improvements towards more robust code and smoother code integration. Also, they comment that the test suite developed helps towards future enhancements and maintenance.

Müller et al. [5] executed a controlled experiment to compare test-first programming (i.e. TDD) with test-last programming. They found that TDD did not increase productivity, and that there was no change in program reliability. However, they did discover that TDD appears to support improved understanding of the code.

In general, research literature divides changes to software into four classes – namely corrective, adaptive, perfective, and preventive. In general, corrective refers to fixing

bugs, and adaptive has to do with new environments or platforms (i.e. evolution). Implementing altered, additional or new requirements, as well as improving performance, can be classified as perfective. Finally, refactoring changes made to improve future maintainability or reuse can be thought of as preventive [12]. Corrective changes can be thought of as software maintenance (i.e. what we here consider defects), while adaptive, perfective and preventive changes can be classified as encompassing software evolution (i.e. which we here call "changes").

Both corrective and non-corrective software changes are a natural part of software maintenance and evolution, respectively. The IEEE definition [16] of software maintenance is as follows: "*Software maintenance is the process of modifying a component after delivery to correct faults, to improve performances or other attributes, or adapt to a changed environment*". On the contrary, there is little agreement on a definition for software evolution in the research literature. Some view software evolution as part of maintenance [12], others view it as a lifecycle step [11]. Belady and Lehman [17] first used the following definition of software evolution: "*....the dynamic behaviour of programming systems as they are maintained and enhanced over their life times.*". Yet another view is that evolution is enhancement regarding functionality and performance between releases [13]. Based on these descriptions, we define software evolution for this study as:

the systematic and dynamic updating in new/current development or reengineering from past development of component(s) (source code) or other artifact(s) to a) accommodate new functionality, b) improve the existing functionality, or c) enhance the performance or other quality attribute(s) of such artifact(s) between different releases [18].

3. Research Design: Motivation, Research Questions and Context

3.1 Motivation

Software evolution and maintenance issues are important research foci, as the changes they encompass comprise a large majority of software development costs (~70%). Nevertheless, these changes cannot be anticipated and thus avoided, since they are required for the ability to modify software towards future needs in a fast and reliable manner. This is the very ability that allows software companies to take advantage of new opportunities and thereby stay competitive [11].

As mentioned, there are few empirical studies on industrial systems using TDD, most of them on the techniques potential for quality improvements. This means that prior investigations have commonly focused on detection and elimination of defects, in relation to general software development [3]. We are studying TDD in a software reuse setting to see whether improvements can be shown by empirical data on Trouble Reports and Change Requests. In comparison with earlier studies we also investigate the relation between defect density and change density to discover potential improvements for the TDD releases, compared to traditional test-last development.

Our aim in this study is to investigate how TDD performs in comparison with prior traditional development of the JEF framework, with respect amount and type of defects and changes, as well as the relation between the two.

3.2 Research Questions

The following research questions have been obtained through a literature survey, and have been adapted towards our use in this investigation:

RQ1: How does defect density for the reusable framework evolve using test-last approach vs. TDD over several releases? Earlier studies on TDD in the software industry have shown decreasing defect density of 40-50 % compared to non-TDD development, but until now only for non-reusable software. Our aim is to investigate whether similar benefits can be seen for reusable components to indicate the relative level of reliability. This is important because such defect reduction has the potential to significantly impact future maintainability in a positive direction. According to Mohagheghi et al. [22] reused components have fewer defects and their requirements are more stable, possibly because they have tougher requirements in terms of predictability, reliability, stability and maintainability. Defects and changes in reusable components also affect the applications reusing them. In this investigation we are comparing the reusable components developed using test-last versus test-driven/test-first development. The aim is to see whether TDD may provide additional improvement towards software reuse, together with these benefits of systematic reuse mentioned in [22].

RQ2: How does change density for the reusable framework evolve using test-last approach vs. TDD over several releases? The change density indicates the degree of enhancements (evolution) the reusable components are subjected to between releases. We here want to see whether the development approach (i.e. test-last vs. test-first) affects the change density. We also wish to see how it contributes to the stability of the reusable components. Change density was not explicitly investigated in prior industrial studies on TDD [3], but is nevertheless important towards characterizing evolution. Furthermore, a prior investigation on reusable components [22] showed that these already have a lower code modification rate (are more stable) than non-reused components using a test-last approach. Here too, we want to investigate potential additional benefits of using TDD, in addition to those that can be achieved through systematic reuse.

RQ3: What is the relation between defect density and change density using test-last approach vs. TDD over several releases? Prior investigations on TDD and non-reusable components reveal decreased defects, but also point towards a possible increase in change density due to lack of design, etc. Here, we explore the relation between defect density and change density for the two last releases (rls4 and rls5, developed using TDD) vs. for the three first releases (rls1, rls2, rls3, developed using a test-last approach) of the reusable JEF framework. The purpose is to see whether such trends can be seen for the reusable components.

3.3 Context

StatoilHydro is a major, multinational oil and gas company. Represented in 28 countries, it has a total of 24,000 employees, with its main headquarters in Europe.

The central IT-department in StatoilHydro develops and releases domain-specific software to achieve better operation flexibility for central business areas of the company. Additionally, they operate and provide support for internal IT-systems in use

within StatoilHydro. The department comprises about 100 developers worldwide, with main locations in Norway and Sweden. Exploring potential systematic reuse benefits has been a key IT strategy of the O&S (Oil Sales, Trading and Supply) business area since 2003. The strategy has been implemented by developing a framework of reusable components called JEF. This framework is based on JEF (Java Enterprise Framework) components, and was developed in response to changing business and market trends.

Another major incentive has been to the ability to use a consistent and resilient technical platform for development and integration of software systems [19]. This reuse strategy is now being propagated to and adopted in other divisions within StatoilHydro ASA. Following the third release of the framework it became apparent that a new focus was needed in order to improve the architecture of the framework and make it more reusable, both in terms of the services provided and the components used. Up to this point, the JEF framework comprised seven different components, which in later releases has been reduced to five. We will therefore be studying the framework on a per release basis in this investigation.

StatoilHydro is part of our industrial cooperation and we are analyzing their data to provide feedback. Two overall change categories are used in StatoilHydro ASA [19]. These are:

- *scope changes:* enhancement/change requests (CR) related to perfective, adaptive and preventive changes, and
- *incidents:* trouble reports (TR) of defects, related to corrective changes. However, an incident can still be classified as a *scope change*, but will then nevertheless be corrected as a defect.

In this article, as aforementioned in section 2, "defects" refer solely to corrective changes, while "changes" refers to enhancement (i.e. perfective, adaptive and preventive; – non-corrective) changes collectively.

When a change request or trouble report is identified by StatoilHydro, it is written and registered in Rational ClearQuest. In this article, we are using both overall categories to investigate whether the company's own experiences with TDD are reflected in terms of change and defect densities. The change requests are the source of changes between releases, while the trouble reports show the defects found between releases, following the first release. In summary, between releases the change requests show experienced evolution, while the trouble reports show needed maintenance. A complete description of change data handling in StatoilHydro ASA is reported in [19]. Data on defect density and change density for the reusable components, using what is here called the test-last approach, were also analyzed and compared to non-reusable components by us in [19].

The latest version of the data was obtained in December 2007. All data was extracted from ClearQuest and exported to Microsoft Excel. The change requests are from 5 releases of the JEF components, releases 1, 2, 3 using traditional test-last development and releases 4 and 5 using TDD. Table 1 shows the size and release date of each of the five JEF releases analyzed in this investigation. The size measure given is the total for each individual release and includes only in-house developed code.

Table 1. The size and release date of the five JEF releases

Using traditional development methodology			Using Test Driven Development	
Release 1: 14. June 2005	Release 2: 9. September 2005	Rls 3: 18. November 2005	Rls 4: 18. April 2007	Rls 5: 11. December 2007
16875 NSLOC	18599 NSLOC	20348 NSLOC	8418 NSLOC	10119 NSLOC

4. Results

All the statistical data presented in this study are based on valid change requests and trouble reports, as none were missing data. Microsoft Excel was used as a tool to analyze changes for the five releases of the reusable JEF framework. In total, there has been 271 (test-last: 223, TDD: 48) recorded trouble reports and 224 (test-last: 206, TDD: 18) recorded change requests, according to release data in Table 1.

RQ1: How does defect density for the reusable framework evolve using test-last approach vs. TDD over several releases? In the versions using traditional development methodology, a total of 10 defects were non-valid. This is because they were either rejected (1), only assigned (3), in progress (3), duplicate (2) or non-fault (1). Furthermore, three of them were noted as duplicates in TDD, and are therefore not used in our analysis here. That is, the 258 (271-10-3) remaining trouble reports were used in our analysis. We first plotted the data using a line plot seen in Figure 1 below. From this figure, we can see that the defect density is decreasing over the three releases using the test-last approach. When using TDD, however, there is a spike in defect density for the first release (rls 4), while TDD also appears to yield a decreasing defect density over the two releases (rls 4 and rls 5) we investigated.

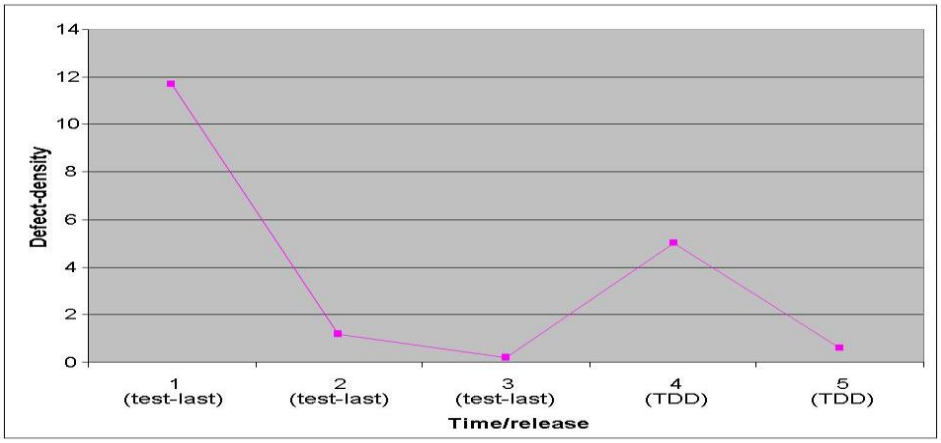

Figure 1. Defect density per release

Table 2 shows the defect density (i.e. number of defects / KNSLOC (noncommented)) for the five releases of the reusable JEF framework investigated. The relative change for release n is (defect density of release n – defect density of release n-1)*100 / defect density of release n-1.

Table 2. Defect density for the JEF framework

Releases	Mean	Defect density	Relative trend
1st	4.35 (of 1st, 2nd, 3rd)	11.67	n/a
2nd		1.18	-89.88%
3rd		0.2	-83.05%
4th	2.79 (of 4th and 5th)	4.99	2395%
5th		0.59	-88.17%

From this table, we see that the defect density of the reusable framework decreases for both approaches, though the first release where TDD was used yields a spike in defect density, as mentioned previously. The mean defect density for the test-last approach was 4.35 ((11.67+1.18+0.2)/3), while for the TDD approach this was 2.79 ((4.99-0.59)/2). This is a relative difference of -35.86%. In terms of mean defect density, we can therefore support the results from prior studies [4][6] that TDD yields fewer defects overall.

RQ2: How does change density for the reusable framework evolve using test-last approach vs. TDD over several releases? Analyzing the changes made to the reusable framework over several releases, 14 were marked as rejected and are therefore not part of this analysis. Consequently, 210 (214-10) change requests were used in our analysis. Again, the data was plotted as a line graph, here seen in Figure 2. This figure shows that the change density for the reusable framework decreases over several releases for the test-last approach. However, for the TDD approach, the change density appears to increase over two releases (rls 4 and rls 5).

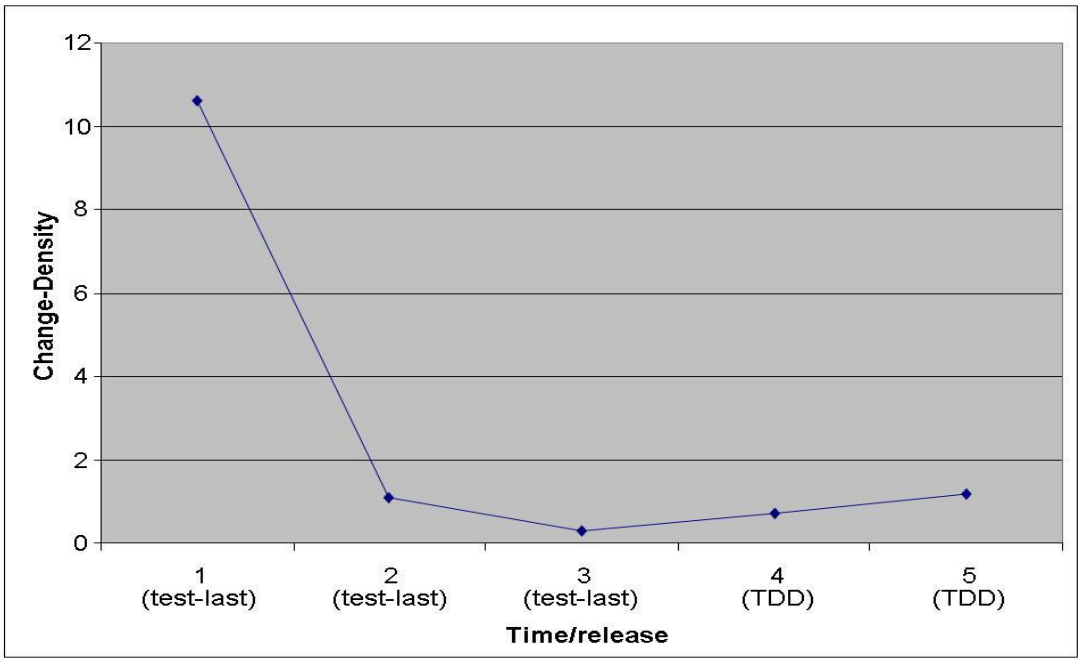

Figure 2. Change density per release

Table 3 shows the change density (#change requests/KSLOC) for each release of the JEF framework. The relative change is again calculated as (change density of release 2 – change density of release 1)*100 / change density of release 1) for each release in relation to the respective corresponding prior release.

Table 3. Change density for the JEF framework

Releases	Mean	Change density	Relative trend
1st	3.99 (of 1st, 2nd, 3rd)	10.61	n/a
2nd		1.08	-89.82%
3rd		0.29	-73.14%
4th	0.95 (of 4th and 5th)	0.71	144.82%
5th		1.19	66.38%

We can see from Table 3 that the change density in the test-last approach starts very high (10.6), followed by a steep drop (-89.82%). The change density for the TDD approach yields an increase over the last test-last release (144.82%), and continues to increase over the next release (rls5: 66.38%). Also, the mean change density for the test-last approach was 3.99 ((10.61+1.08+0.29)/3), while for TDD it was 0.95 ((0.71+1.19)/2). This yields a -76.19% relative difference between the two approaches.

RQ3: What is the relation between defect density and change density using test-last approach vs. TDD over several releases? When it comes to the relation between defect density and change density we used all trouble reports and change requests mentioned under RQ1 and RQ2 above. The graph in Figure 3 shows the plot of the defect density vs. change density for the reusable framework over several releases, distinguishing between the test-last and TDD approaches. From this figure, we can see that for the test-last approach, both defect density and change density decrease over several releases. However, for the TDD approach, the defect density decreases, while change density increases. This is possibly due to refactoring, leading to an increase in preventive changes. To see whether such an increase in preventive changes is the case here, we further classified the changes according to change type (i.e. perfective, adaptive, and preventive).

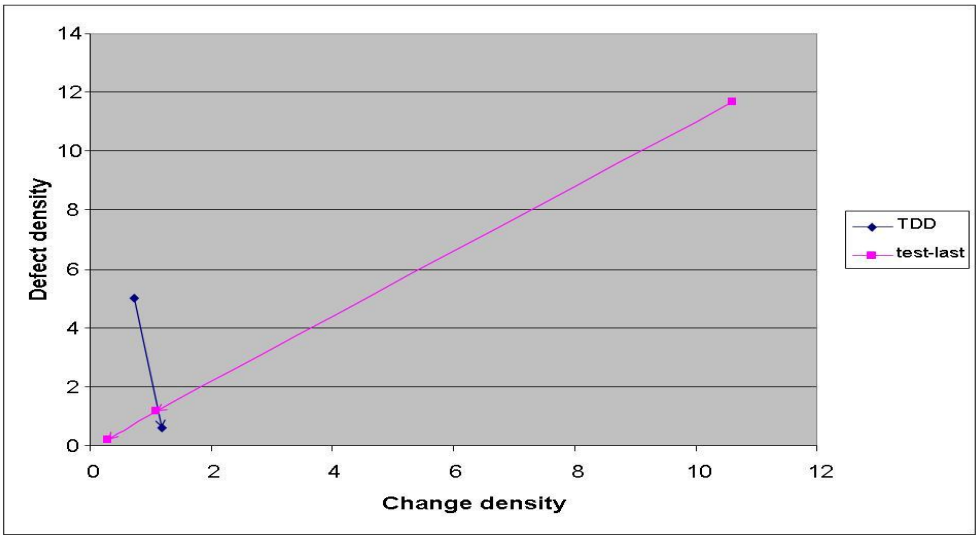

Figure 3. Defect density vs. change density over several releases

The distribution of changes for the test-last approach was analyzed by us in [19]. The results from that study showed the distribution to be 59% perfective, 27% adaptive and 14% preventive changes [19] for the test-last approach. The results from our analysis here show that the distribution of changes for the TDD approach was 33.3% perfective, 5.6% adaptive and 61.1% preventive.

5. Discussion

In Table 6, we have summarized our analysis results, along with corresponding research questions.

Table 6. Summary of the results

Focus	ID	Research Question	Results
Defect density	RQ1	How does defect density for the reusable framework evolve using test-last approach vs. TDD over several releases?	Relative change in mean defect density per release was -35.86% for TDD compared to traditional test-last development (Table 2).
Change density	RQ2	How does change density for the reusable framework evolve using test-last approach vs. TDD over several releases?	Relative change for mean change density per release was -76.19% for TDD compared to test-last development (Table 3).
Relationship: defect density and change density	RQ3	What is the relation between defect density and change density using test-last approach vs. TDD over several releases?	Test-last: Decreasing defect density and change density. TDD: Decreasing defect density, but increasing change density.

5.1 Benefits of using TDD

Neither software evolution nor reusable components were explicitly investigated in prior studies on TDD. Our investigation on reusable components spans five releases, three using test-last and two using TDD. One benefit claimed in earlier studies on TDD is in terms of reduced number of defects on the magnitude of 40-50% [2][4][6] over the traditional test-last approach. In our study, we find that the mean defect density is reduced by 35.86%, so we can support this finding. An obvious consequence of using TDD is that the discovery of more new defects leads to additional test or validation and verification cases being included into the test suites [2]. This means that the "test suite" developed within TDD is developed to become a reusable asset towards testing and discovery [2]. Also, developing the test cases prior to the actual implementation means that defects discovered later are simpler to pinpoint solutions for. Furthermore, the point of refactoring inherent in TDD [2] may also help towards fixing defects. However, the refactoring also increases the number of changes, as shown from our analysis on the distribution of changes under RQ3 above, where the majority of the changes incurred on the TDD approach are preventive.

The trends shown appear to change somewhat over several releases, especially related to change density. The decrease for the mean change density between the two development approaches reflects that the JEF framework has already been in development for several releases prior to the introduction of TDD. Thus the framework's stability in terms of incorporating new requirements is continuing across development methods. Furthermore, this supports the advantages towards code comprehension discussed in [2], and implies that TDD helps towards improving

software reuse. Systematic reuse also offers benefits towards managing software evolution independent of the development approach used [25].

5.2 Drawbacks of using TDD

In terms of drawbacks of using TDD, our results on change density indicate that context (such as reusable vs. non-reusable components and prior knowledge of TDD) plays a large role in writing new test cases, as reported in [2]. Changes (non-corrective) were not explicitly considered in earlier studies on TDD. Though the mean change density is lower, our results show an increasing change density for the TDD approach. Furthermore, the distribution of changes on the TDD approach shows that the majority of the changes made to these releases are preventive changes (61.1%). This is likely due to the increased focus on refactoring inherent to TDD – refactoring is also seen as a partial disadvantage in earlier studies [2] due to the extra time and effort required. We find this interesting, considering that code comprehension is claimed as a benefit in earlier studies [2]. That is, whether there is potentially a larger number of changes, but less effort required per change, using TDD, will remain an issue for further investigation.

One reason for the higher defect and change densities seen in release 4 (the first using TDD) may be that this is the first TDD development performed in the company. This reflects and supports the notion of a learning curve for TDD, indicated in [7][8].

Another reason may be the introduction of new requirements from other systems. We interviewed a senior developer working with JEF on this issue, and the framework is steadily in the process of being propagated to and adopted by other divisions and departments in the company. This means that these systems likely will infer additional new and changed requirements on the framework as it is being further reused and refined. It should also be noted that whether a component is reusable or not, the more it is used, the more changes (and defects) it is likely to incur.

As mentioned, a higher change density may also indicate a higher level of adaptation (i.e. a benefit rather than a drawback). This may be due to an increasing abstraction level, higher number of effective users (i.e. due to the number of dependent components) and middleware-like position (i.e. reusable components often provide communication or security services typically provided by middleware) of the reusable components. All of these characteristics are factors which may influence reusable components independently from the development approach. Though the reusable components we have investigated are all subject to the same influences, these may of course change over time.

Lack of design was claimed as a potential disadvantage of TDD in [2][9][10] towards designing and maintaining the architecture of a system. Although we did not explicitly investigate this issue in our study, developers in the company indicate that they've retained their documentation practices and increased their focus on proper logging together with the introduction of TDD as a new development approach. However, they do not maintain any special documentation related to their use of TDD. Also, their experience is that the usefulness of TDD as a development approach depends on the clarity of the requirements. Where the requirements are unclear, an extra overhead is often needed since the tests have to be updated or rewritten very often, in addition to the code.

5.3 Threats to Validity

We here discuss the possible threats to validity in our survey, using the definitions provided by [20]:

Construct Validity: The analysis constructs we have used (defect density and change density) are based on well-founded concepts in the software evolution and maintenance field. All our data are pre-delivery change requests and trouble reports from the development phases of each new subsequent release. This is similar to the data used in at least one other study [14]. Also, we have made an effort to exclude any invalid or incomplete records for trouble reports or change requests.

External Validity: The object of study is a framework consisting of only five to seven reusable components, and the data has been collected for 5 releases of these components. The framework is currently being reused in two applications within the company. Our results should be relevant and valid for other releases of these components, as well as for comparable contexts in other organizations.

Internal Validity: All of the change requests and trouble reports for the JEF components have been analyzed and the calculations have been performed by us using the tool(s) mentioned. The calculations were double-checked to ensure compliance and correctness, and feel that we've done our utmost to eliminate any possible errors. Also, all the change requests and trouble reports used were complete and valid.

Conclusion Validity: This analysis is performed based on a complete set of data as available at the time the analysis was performed. We therefore think that this data set should be sufficient to draw relevant and valid conclusions.

6. Conclusion and Future Work

We have carried out an investigation on defect and change density in relation to the use of the Test Driven Development vs. test-last approaches on a framework of reusable components. Our results in this study have been presented to StatoilHydro ASA (the origin of the data), and can be summarized as follows:

- We found the relative change in mean defect density per release to be -35.86% for TDD compared to traditional test-last development.
- The relative change for mean change density per release was -76.19% for TDD compared to test-last development.
- The distribution of changes for the TDD approach was 33.3% perfective, 5.6% adaptive and 61.1% preventive.

The results have been presented to Statoil ASA and contribute towards understanding the implications of using TDD as a development methodology, as well as possible impacts of switching development methods. The results will also be combined with other research in the company to explain findings regarding effort and reuse. In this way, they hope to use this work as input towards improving current and future reuse programs at StatoilHydro. Additionally, we plan to expand our dataset to include additional releases of the reusable framework, and to refine the research questions based on our findings here. Future work includes an investigation of effort towards defects (maintenance) and changes (evolution) to investigate other potential facets of benefits related to TDD.

7. Acknowledgements

This research was initiated through results from prior interviews on software engineering by the software engineering group, in the department of information and computer science at NTNU. We thank all parties involved, and especially the company for allowing us access to and use of their data in our research. The study was performed in the SEVO (Software EVOlution in component-based software engineering) project [21], which is a Norwegian R&D project in 2004-2008 with contract number 159916/V30.

8. References

[1] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of test-first approach to programming", *IEEE Trans. Sw. Engr*, vol. 31(1), p.1–12, January 2005.

[2] B. George and L. Williams, "A structured experiment of test-driven development", *Information and Software Technology*, vol. 46(5), p. 337–342, 2004.

[3] D. Janzen and H. Saiedian, "Test-driven development: concepts, taxonomy and future directions", *IEEE Computer*, 38(9):43–50, Sept 2005.

[4] E. M. Maximilien and L. Williams, "Assessing test-driven development at IBM", *Proc. ICSE'2003*, p. 564–569, Piscataway, NJ, May 3–10, 2003.

[5] M. Müller and O. Hagner, "Experiment about test-first programming", *IEEE Proc. Software*, 149(5):131–136, 2002.

[6] L. Williams, E. Maximilien, and M. Vouk, "Test-driven development as a defect-reduction practice", *Proc. 14th IEEE International Symposium on Software Reliability Engineering*, p. 34–45, Nov. 2003.

[7] A. van Deursen, "Program comprehension risk and opportunities in Extreme Programming", *CWI*, Amsterdam, SEN-R0110, ISSN 1386-369X, 2001.

[8] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code", *Proc. 2nd Int'l Conf. on Extreme Programming and Flexible Processes in Sw. Engr. (XP2001)*, pp. 92-95, University of Cagliary, 2001.

[9] B. Foote and J. Yoder, "Big ball of mud", *4th Conference on Patterns, Languages of Programs*, Monticello, Illinois, September 1997.

[10] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture", *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52, October 1992.

[11] K. H. Bennett and V. Rajlich, "Software Maintenance and Evolution: A Roadmap", *Proc. ICSE'2000 – Future of Software Engineering*, Limerick, Ireland, ACM press, pp. 73-87, 2000.

[12] I. Sommerville, *Software Engineering*, Seventh Edition, Addison-Wesley, 728 p., 2004.

[13] P. Mohagheghi and R. Conradi, *An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes*, ISESE 2004, Redondo Beach (Los Angeles), USA, 19-20 Aug. 2004.

[14] P. Mohagheghi and R. Conradi, "An Empirical Study of Software Change: Origin, Acceptance ratet, and Functionality vs. Quality attributes", *ISESE 2004*, Redondo Beach (Los Angeles), USA, 19-20 Aug. 2004, pp. 7-16.

[15] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Second Edition, Addison-Wesley, 2004.

[16] IEEE Std. 1219: *Standard for Softare Maintenance*, Los Alamitos, IEEE Computer Society Press, CA, USA, 1993.

[17] L. A. Belady and M. M. Lehman, "A model of a Large Program Development", *IBM Systems Journal*, 15(1):225-
252, 1976.

[18] O. P. N. Slyngstad, J. Li, R. Conradi, and M. Ali Babar, "Identifying and Understanding Architecural Risks in Software Evolution: An Empirical Study", *Proc. Profes 2008*, 15 p., to appear.

[19] A. Gupta, O. P. N. Slyngstad, R. Conradi, P. Mohagheghi, H. Rønneberg, and E. Landre, "A Case Study of Defect density and Change density and their Progress over Time", Proc. CSMR'2007 – Software Evolution in Complex Software Intensive Systems, p. 7-16, 21-23 March 2007, Amsterdam, The Netherlands, IEEE CS Press.

[20] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in Software Engineering – An Introduction*, Kluwer Academic Publishers, 2002.

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

[22] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz, "An Empirical Study of Software Reuse vs. Defect Density and Stability", *Proc. ICSE'2004*, pp. 282-292, IEEE-CS Press. Ext. version in *ACM Trans. on Sw. Engineering and Methodology (TOSEM)*, July 2008, 34 p., to appear

[23] D. Hamlet and J. Maybee, *The Engineering of Software*, Addison-Wesley, Boston, 2001.

[24] W.S. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, MA, 1989.

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

P5: Identifying and Understanding Architectural Risks in Software Evolution: An Empirical Study

Published in proceedings of PROFES'2008.

Odd Petter Nord Slyngstad[1], Jingyue Li[1], Reidar Conradi[1], M. Ali Babar[2]

[1] Department of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU), Trondheim, Norway
{oslyngst, jingyue, conradi}@idi.ntnu.no
[2] LERO– The Irish Software Engineering Centre, University of Limerick, Limerick, Ireland
malibaba@lero.ie

Abstract. *Software risk management studies commonly focus on project level risks and strategies. Software architecture investigations are often concerned with the design, implementation and maintenance of the architecture. However, there has been little effort to study risk management in the context of software architecture. We have identified risks and corresponding management strategies specific to software architecture evolution as they occur in industry, from interviews with 16 Norwegian IT-professionals. The most influential (and frequent) risk was "Lack of stakeholder communication affected implementation of new and changed architectural requirements negatively". The second most frequent risk was "Poor clustering of functionality affected performance negatively". Architects focus mainly on architecture creation. However, their awareness of needed improvements in architecture evaluation and documentation is increasing. Most have no formally defined/documented architecture evaluation method, nor mention it as a mitigation strategy. Instead, problems are fixed as they occur, e.g. to obtain the missing artefacts.*

Keywords: software architecture, software evolution, risk management, software architecture evaluation

1. Introduction

Modern software systems are commonly built by acquiring and integrating various components developed by commercial or open source entities. The software engineering community has enabled several processes for developing and maintaining component-based systems. Proper handling of software architecture is one of the most important factors towards successful development and evolution of component-based systems. However, there has been little effort to identify and understand the architectural risks in software evolution and potential strategies to deal with those risks. We assert that it is important to obtain and disseminate the information about potential risks (i.e. problems) in architecture evolution, as the architecture constitutes the central part of a software system [1]. Knowledge and understanding about architecture evolution risks should facilitate the development of improved strategies to mitigate these risks.

We have decided to obtain such knowledge from practicing IT-professionals working with software architecture, as they are expected to encounter risks (i.e. problems that may occur) in evolving software architectures on a regular basis. Our research here is concerned with Component-Based Software Engineering (CBSE) development, where there has been architectural evolution during the systems' lifetime.

Using a convenience sample of respondents, we carry out a preliminary investigation of architectural risks and management strategies in software evolution. This means changes to the structure(s) of a system of software elements, their external properties and mutual relationships, all viewed from a perspective of risk analysis and risk

mitigation. This exploratory study is targeted at Norwegian IT-professionals who hold significant knowledge and experience in designing and evolving software architectures.

We have identified architectural risks (i.e. problems identified in planning or experienced during the maintenance/evolution) and associated risk management strategies (i.e. methods to mitigate these issues) as they occur in industry. "Lack of stakeholder communication affected implementation of new and changed architectural requirements negatively" was the most influential as well as the most frequent risk. This risk was most effectively mitigated by extending the time used towards communication with stakeholders. "Poor clustering of functionality affected performance negatively" was the next most frequent risk. This risk was in turn most successfully mitigated by refactoring or improving the modifiability of the architecture.

Furthermore, architects easily handle anticipated or experienced risks. However, their focus is usually on "forward engineering", not on reengineering (i.e. the architecture solution rather than the suitable steps to get there [14] in advance). Despite this, some of the findings also show that awareness of software documentation and evaluation issues and practices is increasing. Also, most of the respondents have no formally defined or documented architecture evaluation method in place. Rather, challenges are met as they appear, and the main focus is on obtaining the missing artifact. Finally, none of our respondents mentioned using formally defined or documented architecture evaluation as a risk mitigation strategy.

The remainder of the paper is organized as follows: Section 2 holds Background. Research Design is in Section 3. Section 4 contains information on our data collection, and the results of our study are in Section 5. Discussion and Threats to Validity are located in Section 6, and Conclusions and future work are in section 7.

2. Background and Related Work

Software Architecture [1] can be defined as the discipline dealing with the structure or structures of a system, comprising software elements, the externally visible properties ("interface" of in-going and out-going calls) of those elements, and the relationships between them. Well-defined software architecture is one of the key factors in successfully developing and evolving a non-trivial system or a family of systems. A well-defined software architecture provides a framework for the earliest design decisions to achieve functional and quality requirements. In addition, it has a profound influence on project decomposition and coordination. Poor architecture often leads to project inefficiencies, poor communication, and inaccurate decision making [1]. The above definition of software architecture refers to software elements, which can be seen as components of the given software system. Hence software architecture is closely related to CBSE [2].

Clerc et. al. [14] conducted a study to understand architects' attitudes towards software architecture knowledge. They found that architects are aiming more at creation and communication instead of review and maintenance of a system's architecture. Bass et al. [21] analyzed the output from 18 ATAM evaluations to discover risk themes specifically for software architecture. Besides a set of risk categories, they found that the more prevalent risks are those of omission (i.e. of not taking action on a particular issue). They also did not find a link between the risk categories and the business/mission goals or the domain of a system. Bass et al. further comment that the

similarities to their study shown in [23] indicate the industrial relevance of the risk categories [21], as well as the ability of ATAM analysis to discover architectural risks. Another risk categorization from ATAM evaluations is presented in O'Connell [22], using 8 evaluation results. Although study [22] was analyzed independently from [21], the resulting themes are similar in content. It should be noted though that neither of these studies deal explicitly with the evolution of software architecture.

The architecture of a system will evolve as architectural changes are accumulated over time. There are diverging views in the research community about how software evolution should be defined. These include considering maintenance as a broader term [5], seeing evolution as a step in the software lifecycle [4], and regarding evolution as software systems' dynamic behavior through maintenance and enhancements [3]. Some [9] consider evolution as the enhancement and improvement performed on a system between releases. Based on this description, we define software evolution for this study as: *the systematic and dynamic updating in new/current development or reengineering from past development of component(s) (source code) or other artifact(s) to a) accommodate new functionality, b) improve the existing functionality, or c)enhance the performance or other quality attribute(s) of such artifact(s) between different releases.*

If left unchecked, over time, a system's architecture will naturally decay as new quality and functional requirements are imposed on it. This decay is manifested by the original architectural structure(s) being lost. This is sometimes called "software rot" [20], and is one of the most prevalent reasons behind reengineering the architecture of a software system.

Risk management entails methods to mitigate risks that may occur during a software development project. Boehm [8] describes a framework for risk management consisting of two main steps, namely risk assessment (identification, analysis, and prioritization) and risk control (planning, resolution, and monitoring). Ropponen and Lyytinen [6] have identified six elements of software risk. Their results reveal influence on risk elements by environmental factors (e.g. development method). Also, awareness of risk management importance and method(s) was shown to have an effect. Keil et al. [10] conducted a risk management survey of project managers. They identified several additional important risk factors in comparison with Boehm [8], contributing these to changes in the industry since Boehm's study. Additionally, they discovered that important risks were commonly out of managers' control. They therefore suggested that project managers widen their attention beyond traditional software risk factors.

Further based on the definition of risk in Boehm's article[8], as well as input from [6][12], we use the following definition for architectural evolution risks: *the issues or problems that can potentially have negative effects on the software architecture of a system as it evolves over time, hence compromising the continued success of the architecture*. The above studies on architectural risks [21][22] have focused on discovering risk categories directly from the output of ATAM [1] analyses. They use analysis outputs from organizations where such evaluation is an established practice. However, they do not comment on how commonly such formal evaluation methods are used in industry. Nor do they take software evolution specifically into account. In [7], the authors found that evaluation practices could range from completely ad-hoc to formally planned, from qualitative to quantitative. They also discovered that the approach depended on the goals of the evaluation. This means that additional risk issues and management strategies could be left undiscovered by looking only at output from

structured analysis reports. We therefore decided to employ semi-structured interviews to gather qualitative information on risk issues and risk management strategies.

3. Research Design: Context, Motivation and Research Questions

We observed that risks and risk management strategies are commonly studied in relation to general software development [11][12][13], identifying risks on the project level [6][8][10]. Similarly, software architecture studies often focus on the design, implementation and maintenance of the architecture. While these results are important, there has been little effort to study risk management in the context of software architecture [21][22]. Hence, we decided to carry out an empirical study to help further identify and better understand the risks and risk management strategies in relation to software architecture.

This research is limited to those software systems which have two major characteristics: use of CBSE and changes in the systems' software architectures during their lifetimes. This means projects that have at least delivered the first production release, i.e. can be said to be in the "maintenance" phase.

Our main motivation is to obtain insight into the actual risks (i.e. issues identified and experienced which may affect the software architecture negatively) and associated risk management strategies (i.e. effective mitigation methods), as they occur in industry, in relation to software architecture evolution. We aim to use the results from this exploratory study as basis for more in-depth studies in this area.

This study is aimed at identifying and understanding risks and strategies relevant to software architecture evolution. That is, we investigate the steps of risk identification, analysis and prioritization, as well as risk planning and resolution [8], as they occur in industry. We do not cover issues pertaining to risk assurance or monitoring [8]. The research questions are as follows:

RQ1: What are the relevant architectural risks of software evolution, i.e. what software architecture related risks can be encountered during software evolution? Any issue that can affect a project adversely if not handled correctly is considered a risk [8]. The first step in Boehm's risk management framework [8] entails risk identification, analysis, and prioritization. We are hence here interested in investigating the state-of-the-practice regarding risk awareness, i.e. to obtain insight on which risks that software architects deem more important in relation to software architecture evolution.

As aforementioned, software architecture is the central part of a software system [1], so failure of the software architecture can easily cause the entire project to fail. Hence a proper focus on the software architecture is needed to ensure the project is kept on budget and schedule. Similarly, changes to the software architecture can cause subsequent changes in many components of a software system [1]. It is therefore imperative to be aware of the possible risks incurred on the software architecture through software evolution.

RQ2: How can these risks best be assessed; through which methods or mechanisms were these risks identified, analyzed and prioritized? Software architecture evaluation is widely known as an important and effective way to assess architectural risks [1, 7]. In order to identify, analyze and prioritize [8] risks there is the need for effective methods or mechanisms for software architecture evaluation. Such mechanisms help validate architecture design decisions with respect to required quality

attributes (such as testability, availability, modifiability, performance, usability, security etc.). Prior architecture analysis studies [21][22] focused on structured analysis outputs as a method to discover risks. However the analysis methods used can range quite widely [7]. Investigating a wider range of analysis methods will help discover risk issues possibly missed by earlier studies.

RQ3: How can these risks best be mitigated: what were the relevant risk management strategies? Were the strategies successful or not? The second step in Boehm's framework [8] encompasses risk control. This step focuses on problem mitigation; it is aimed at handling problems to minimize their impact. Here, our aim is to obtain the status quo, and suggest possible improvements by enabling a systematic approach to architectural risk management in software evolution. It is therefore imperative that we receive information on both positive and negative aspects of employed risk management strategies, and also on their outcomes.

Again, risks in relation to the central part of a software system (i.e. the architecture [1]) are important. Proper management of these risks on the three levels, technical, process and organization [11][12][13], provides the ability to minimize the potentially far-reaching impacts of these risks [8].

In order to practically explore the three research questions above, we designed an interview guide consisting of six questions. The relation between the questions in the interview guide, the research questions, and Boehm's framework [8] is shown in Table 1.

Table 1. Relation between research questions and the interview guide

	Identification, Analysis, and Prioritization [8]	Assess-ment [8]	Planning, and Resolution [8]
Questions in the interview guide	RQ1	RQ2	RQ3
Q1.1. Describe architectural problems (indicate influence) and strategies (rate outcome) you identified in planning maintenance/evolution?	X		X
Q1.2. Describe architectural problems (indicate influence) and strategies (rate outcome) experienced and employed during maintenance/evolution?	X		X
Q2. Indicate weighting of and any changes in the following quality attributes[1]: testability, availability, modifiability, performance, usability and security) in your software architecture?	X		
Q3. How has the architecture changed throughout the lifetime of the system?	X		
Q4. Please describe your architecture change process?		X	X
Q5 Which architectural patterns (e.g. layering, task control, AI approach pipe-and-filter etc.) did you use to design the architecture?	X		
Q6. Does your organization use a defined and/or documented method or process to evaluate software architecture?		X	X

Question Q6 has been adapted from an earlier empirical study aimed at identifying the factors that can influence software architecture evaluation practices [7]. We also gathered demographic data (e.g. level of experience) about the respondents. The interview guide was piloted with 3 researchers to ensure quality and ease of understanding, through which the questions were polished and refined. We aimed to be

flexible so as to gain as much qualitative information on each question as possible. Therefore, all the questions (Q1-Q6) were left open-ended. Also, the influence of each risk and the outcome of each strategy were indicated on a 5-point Likert scale. That is, risk Influence was ranked Very High = 5 to Very Low = 1. Similarly, strategy Outcome success was ranked Completely = 5, Mostly = 4, Medium = 3, Somewhat = 2 and Not at all = 1 successful.

4. Data Collection and Analysis

This study was carried out using a convenience sample of participants from the software industry in Norway. Potential respondents were first contacted by email, and sent the invitation letter with interview guide to get an overview. Later the potential respondents were contacted again by phone and signed up for a phone-interview appointment if they agreed to participate. The respondents were 16 IT-professionals in different companies with prior knowledge and experience with software architecture.

The phone interviews took on average 30 minutes to carry out, and we obtained complete responses to all the six questions from all 16 respondents. The data was recorded on paper and transcribed into electronic form. The responses were also summarized and read back to the respondents directly after the interviews, so they could be checked for accuracy.

Nine of the respondents had bachelor level degrees, while seven had master degree level educations. On average, the respondents had 8 years of experience working with software architecture, with six having less than five years of experience, five having 5-10 years of experience and another five having over 10 years of experience

We analyzed the data as follows: The data was initially analyzed by dividing the data into discrete parts and coding each piece according to risk or strategy theme(s). As an example, for risks this was done as {condition – what may go wrong, consequence(s)}: e.g. "requirements from earlier versions still in effect affected architecture design negatively." was coded as {earlier version requirements, negative for architecture design}.

We then examined them for commonalities and differences, and grouped related pieces of information based on their coding (e.g. for risks, {earlier version requirements, negative for architecture design} and {required same functionality as before, negative for planning} were grouped as {required backward compatibility, negative for architecture maintenance/evolution planning and design}). Each respondent's transcript was run through this procedure. The results were checked by a second researcher to ensure reliability. This is similar to the constant comparison method described in [16]. The issues identified in the data analysis were classified into three categories; technical, process and organizational. We believe that risk management is not merely a technical issue; rather, it spans all three categories [11][12][13][21].

5. Results

The results are here divided into categories of (1) technical, (2) process and (3) organizational risks. This means that we have combined the findings from Q1.1 and Q1.2 for RQ1 and RQ3.

Technical risks: Table 2 shows the most influential technical risks and corresponding management strategies performed. From Table 2, we can see that the strategy applied in planning towards TR1 was Completely successful (Outcome = 5). Furthermore, overall the strategies were also 3 out of 5 of Medium success (Outcome = 3), and 1 out of 5 Not at all successful (Outcome = 1).

Table 2. Most **influential (Influence ≥ 4) technical** risks (TRs) and corresponding management strategies performed

Technical	ID	Risk	Influence	Strategy	Outcome
Identified in planning	TR 1	Poor clustering of functionality affected performance negatively	4	Refactoring of the architecture	5
Experienced during	TR 2	Poor original core design prolonged the duration of the maintenance/ evolution cycle	4	Improve modifiability of the architecture	3
	TR 3	Increased focus on modifiability contributed negatively towards system performance	4	Implementation of changes towards modifiability	3
	TR 4	Varying release cycles for COTS/OSS components made it difficult to implement required changes	4	Use own development as potential backup	3
	TR 5	Poor clustering of functionality affected the performance negatively	4	Implement extra architecture add-ons	1

Process risks: Table 3 (below) shows the most influential process risks and corresponding management strategies performed. These results (Table 3) show that all of the strategies used in response to the most influential risks in planning were Completely successful. Towards the risks experienced during the maintenance/evolution, the strategies were 3 out of 10 Completely successful, 5 out of 10 of Medium success, while 2 out of 10 were Completely successful.

Table 3. Most **influential (influence ≥ 4) process** risks (PRs) and corresponding management strategies performed

Process	ID	Risk	Influence	Strategy	Outcome
Identified in planning	PR1	Lack of architecture documentation contributed to more effort being used on planning the maintenance/ evolution	4	Recover arch. documentation from current architecture design	5
	PR2	Lack of architecture evaluation delayed important maintenance/ evolution decisions	4	Recover evaluation artefacts where needed	5
				Alter process to capture important details	5
Experienced during	PR3	Lack of stakeholder communication affected implementation of new/ changed architectural requirements negatively	5	Negotiated project extension	3
				Allow additional time for communication/feedback	5
	PR4	Insufficient requirements negotiation contributed to requirement incompatibilities on the architecture	4	Postponed some requirements to next maintenance/evolution cycle	3
	PR5	Poor integration of architecture changes into implementation process affected implementation process and the architecture design negatively	4	Overlay new architecture change process onto implementation process	5
				Integrate architecture considerations into implementation process	3
	PR6	Using Software Change Management (SCM) sys. w/o explicit software architecture description contributed to inaccuracies in communicating the architecture	4	Use separate system for architecture description (using ADL), link to SCM system	3
				Trial use of additional ADL system	3
	PR7	No standard terminology affected internal and external communication efforts negatively	4	Align terminology with literature	1
				Extra communication to clarify terminology	1
	PR8	Customer architects being unfamiliar with architecture change process affected maint./ evo. cycle schedule negatively	4	Extra communication effort with own resident architect to clarify	5

Organizational risks: Table 4 (below) shows the most influential organizational risks and corresponding management strategies performed. Among the strategies used in response to these most influential organizational risks (Table 4) identified in planning, 2 out of 4 were Medium successful, while 2 out of 4 were Completely successful. Towards those experienced during, the strategies were all Medium successful.

Table 4. Most **influential (influence ≥ 4) organizational** risks (ORs) and corresponding management strategies performed

Organization	ID	Risk	Influence	Strategy	Outcome
Identified in planning	OR 1	Architecture team on a per maintenance/evolution cycle basis contributed to loss of knowledge about the existing architectural design	4	Dedicated personnel to "retrieve" knowledge	3
	OR 2	Cooperative maintenance / evolution with architects from customer organization required extra training and communication efforts	4	Frequent, interactive, scheduled meetings to keep up to date	5
	OR 3	Lack of clear point of contact from customer organization contributed to inconsistencies in communication of the architecture and requirements	4	Involve all "layers" of customer organization as stakeholders, allow extra communication time	5
	OR 4	Not allowed to change OSS as decision mandate external to architecture team, affecting performance and modifiability negatively	4	Ensure compliance with external mandate holder	3
Experienced during	OR 5	Separate architecture team per maint. / evo. cycle contributed to insufficient knowledge about the existing architectural design	4	Regain architecture details from upper management remaining	3
	OR 6	Prior architecture maint./ evo. by other projects due to lack of personnel made it difficult to obtain existing architecture design documentation	4	Merge architecture knowledge / documentation to central location	3
	OR 7	Large architecture team affected division of duties and subsequently implementation of maint./ evolution cycle negatively	4	Divide duties between subgroups	3
	OR 8	Lack of clear lead architect affected implementation progress negatively and contributed to extra effort needed	4	Merge duties and diverge roles more clearly	3

Additionally, our results show that the overall most frequent (and most influential) risk was "Lack of stakeholder communication affected implementation of new and changed architectural requirements negatively". The most successful strategy in response to this risk was "Allow additional time for communication for communication and feedback". The second most frequent risk was "Poor clustering of functionality affected performance negatively", with "Refactoring the architecture" and "Improve the modifiability of the architecture" as corresponding most successful strategies. The results from questions Q2, Q3, Q5 (Table 5), and Q4, Q6 are below.

Table 5. Summary of additional findings for RQ1

<table>
<tr><td colspan="2">Q2. Quality attribute foci:</td></tr>
<tr><td colspan="2">• Focus on any given QA can change during the project.
• Only a few projects experienced a lowering of focus on a given QA.
• Most frequent QA with increased focus was Modifiability, followed by Usability.</td></tr>
<tr><td colspan="2">Q3. Architecture changes made during system lifetime to:</td></tr>
<tr><td>• Improve processing speed or scale (7 out of 16)
• Improve flexibility to accommodate future changes (7 out of 16)
• Accommodate new or altered user requirements (5 out of 16)</td><td>• Improve system uptime (3 out of 16)
• Enable additional access interfaces (1 out of 16)
• Increase abstraction level (1 out of 16)
• Support additional record types (1 out of 16)</td></tr>
<tr><td colspan="2">Q5. Architectural patterns used (as means to solve design challenges):</td></tr>
<tr><td>• Inversion of Control (1 out of 16),
• Layered (3 out of 16),
• Blackboard (3 out of 16),</td><td>• Model View Controller (4 out of 16),
• Pipeline (3 out of 16),
• Task Control (2 out of 16), and
• Broker (1 out of 16).</td></tr>
</table>

The following are results from Q4 (**RQ2, RQ3**) (architecture change process):

- none used a strictly defined change process,
- 7 out of 16 performed this process informally,
- 4 out of 16 employed loosely defined procedures,
- 3 out of 16 changed the architecture as part of the development process, and
- 2 out of 16 just change the architecture as needed.

In question Q6, none of the respondents answered that they have a defined or documented process for software architecture evaluation. 5 out of 16 of the respondents have a loosely defined process in place if needed. Another 5 out of 16 have knowledge of evaluation processes or methods mentioned in literature. Yet another 5 out of 16 of the respondents carry out a software architecture evaluation informally if needed. Finally, 1 out of 16 of the respondents reports that her/his organization has a process for software architecture evaluation in place (in this specific case, based on the Architecture Tradeoff Analysis Method – ATAM [1]), but this is not commonly used.

6. Discussion

6.1 Comparison to Related Work

The Technical risks identified by the respondents show a high focus on design and creation of the architecture, supporting [14].

While Ropponen's [6] focus was overall software development risks, ours is software architecture risks in software evolution. The strategies used in response to the risks we identified as (See Table 6 below) "Architecture Team" and "Requirements" risks were reported as being Medium or Completely successful in outcome. We can hence support the notion that there is at least some success in managing risks related to "Architecture Team" and "Requirements" [6].

A summarized comparison with the above and Bass et al. [21] is also in Table 6.

Table 6. Summary of comparison to related work

ID	Ropponen et al. [6]
	Requirements risks:
PR4	"Insufficient requirements negotiation contributed to requirement incompatibilities"
TR3	"Increased focus on modifiability contributed negatively towards system performance"
	Architecture Team risks:
OR5	"Separate architecture team per maint. / evo. cycle contributed to insufficient knowledge about the existing architectural design"
OR7	"Large architecture team affected division of duties and subsequently implementation of maint./ evo. cycle negatively"
OR8	"Lack of clear lead architect affected implementation progress negatively and contributed to extra effort needed"
	Stakeholder risks (from the subcontractor viewpoint):
PR3	"Lack of stakeholder communication affected implementation of maint./ evo. cycle negatively"
OR2	"Cooperative maint./evo. w/ architects from customer organization required extra training and communication efforts"
OR3	"Lack of clear point of contact from customer organization contributed to inconsistencies in communication of the architecture and requirements"
PR8	"Customer architects being unfamiliar with architecture change process affected maint./evo cycle schedule negatively"
ID	**Bass et al. [21]**
	Quality Attribute risk:
TR3	"Increased focus on modifiability contributed negatively towards system performance"
	Integration risks:
TR4	"Varying release cycles for COTS/OSS components made it difficult to implement required changes"
OR4	"Not allowed to change OSS as decision mandate external to architecture team, affecting performance and modifiability negatively"
	Requirements risks:
PR4	"Insufficient requirements negotiation contributed to requirement incompatibilities on the architecture"
TR3	"Increased focus on modifiability contributed negatively towards system performance"
	Documentation risks:
PR1	"Lack of architecture documentation contributed to more effort being used on planning the maintenance/evolution"
PR6	"Using Software Change Management system w/o explicit software architecture description contributed to inaccuracies in communicating the architecture"
	Process and Tools risks:
PR2	"Lack of architecture evaluation delayed important maintenance/evolution decisions"
PR6	"Using Software Change Management system w/o explicit software architecture description contributed to inaccuracies in communicating the architecture"
	Allocation risks:
TR1	"Poor clustering of functionality affected performance negatively"
TR4	"Varying release cycles for COTS/OSS components made it difficult to implement required changes"
	Coordination risks:
PR3	"Lack of stakeholder communication affected implementation of maint./evo. cycle negatively"
PR8	"Customer architects being unfamiliar with architecture change process affected maint./evo cycle schedule negatively"
OR2	"Cooperative maint./evo. with architects from customer organization required extra training and communication efforts"
OR3	"Lack of clear point of contact from customer organization contributed to inconsistencies in communication of the architecture and requirements"
OR4	"Not allowed to change OSS as decision mandate external to architecture team, affecting performance and modifiability negatively"

6.2 Observations on Key Architectural Risks and Promising Risk Management Strategies

The most influential Process risks we identified (Table 3) show that the **main focus is still forward thinking (producing systems according to budget and schedule) rather than hindsight reflection and learning.** Further, from the answers to Q5 we can see that the consequences of using one or more specific patterns are neither explicitly considered, nor evaluated as potential risks (though tactics, packaged by patterns, is a risk issue also discovered from ATAM reports in [21][22]).

The answers from Q4 and Q6 also point towards this main focus. Hence there is no apparent specific focus on discovering potential problems (rather problems are fixed as they are encountered, focussing on the missing artefacts). This is despite the potential benefits (e.g. identifying architecture design errors and potentially conflicting quality requirements early) of defined and documented architecture evaluation described in the literature [1]. However, architects are **becoming aware that their practices around evaluation and documentation need improvement.** This is echoed by the Organizational risks we identified (Table 4), such as "Architecture team on a per maint./evo. cycle basis contributed to loss of knowledge about the existing architectural design" and "Large architecture team affected division of duties and subsequently implementation of maint./evo. cycle negatively".

A **link to Business Risks** [19] (i.e. those that affect the viability of a software system) can also be seen. The architectural risks identified are influenced by and in turn also affect such elements as e.g. cost, schedule.

Considering the most influential **Technical risks** (table 3), we can see that the majority of them were experienced during the maintenance/evolution, without prior planning. The same appears the case for the most influential **Process risks,** whereas for the most influential **Organizational risks** half were identified in planning, and another half were experienced during the maintenance/evolution. In terms of management strategies, one overall trend appears to be that those employed in response to risks identified in planning had a more successful outcome. This appears especially to be the case where the same risk was both identified in planning as well as experienced during the maintenance/evolution (e.g. Technical risks TR1 and TR5: "Poor clustering of functionality affected performance negatively"). These findings also emphasize the points about forward engineering and awareness discussed above.

One of the strategies applied towards technical risks, as well as two of the strategies applied towards process risks were Not at all successful. These strategies should be viewed in light of the respective projects' context (Tables 2, 3). Additionally, improvement is needed in the employed strategies, especially regarding issues encountered during maintenance/evolution. The lack of a strictly defined and documented architecture change process reported by the respondents (Q4) is also an interesting finding. We would expect architecture evaluation to be part of a given change process in order to analyze the consequences of proposed architectural changes.

To improve this situation, we believe that rigorous documentation and evaluation of architecture should be made an integral part of a software architecture change process. Furthermore, management of risks specific to architectural modifications should be given more attention. To achieve these objectives, software architects should be provided appropriate training. Moreover, organizational management should also

demonstrate commitment to implement changes to the way software architecture changes are handled.

6.3 Threats to Validity

Threats to validity (using definitions provided by Wohlin et al. [15]):

Construct Validity: The research questions are rooted firmly in the research literature, and the actual questions in the interview guide have direct relations to the research questions. The interview guide was refined through pre-testing among our colleagues to ensure quality. All the terms used in the guide were defined at the beginning to avoid any potential misinterpretations.

External Validity: This study has been conducted by using a convenience sample of 16 IT-professionals, an issue which remains a threat. Nevertheless, obtaining a random sample is almost unachievable in software engineering studies because our community lacks good demographic information about populations of interest [17]. The respondents were chosen by us based on their background and experience with software architecture. Each respondent nevertheless represents a different company.

Internal Validity: The respondents are all knowledgeable and from the software industry, and have expressed an interest in the study. They all have the needed knowledge and background to provide informed answers. We hence believe that they have answered the questions to the best of their ability, truthfully and honestly, drawing on their own experiences, skills and knowledge. We also clarified any ambiguities in the questions or the accompanying definitions during the actual interviews, in addition to the definitions provided in the guide.

Conclusion Validity: This is an exploratory study. The findings are based on analyzing data from a relatively small number of software architects. We plan to implement a large scale study to confirm the results of this study. However, the exploratory nature of the study has identified several issues that may cause architectural risks for evolving systems. The insights gained will also function as background for refining the interview guide towards expansion of the sampling base for the planned larger scale study.

7. Conclusion and Future Work

We conducted phone-based, semi-structured interviews of 16 software architects from Norway for an exploratory study regarding risks and risk management strategies occurring in industry related to software architecture evolution.

Our findings include an initial identification of risks and corresponding risk management strategies as they occur in industry. Our main observations include that “lack of stakeholder communication affected implementation of new and changed architectural requirements negatively” was the most influential and frequent risk. The corresponding most successful strategy was to “Allow additional time for communication and feedback”. In second place concerning most frequent risks came “Poor clustering of functionality affected performance negatively”. The most successful management strategies towards this risk were “Refactoring the architecture”, and “Improve the modifiability of the architecture”.

Furthermore, architects' main concerns are towards designing and creating the architecture. However, our results also show some awareness towards improvements in relation to how these tasks are performed, as well as towards the importance of retaining knowledge about and performing evaluation of the architecture. As most respondents have no formally defined or documented method to evaluate software architecture, problems are fixed as they occur with focus on the lacking artefacts rather than on the method.

Our results here will be used as input for a larger study in the software industry to survey the state-of-practice on risk and risk management regarding software architecture evolution. In particular, we plan to explore the relation between risks and risk management practices, and project context factors.

Acknowledgements

We thank all parties involved. The study was performed in the SEVO project, a Norwegian R&D project in 2004-2008 with contract number 159916/V30.

References

1. L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Second Edition, Addison-Wesley, 2004.
2. L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau, *Volume I: Market Assessment of Component-based Software Engineering* in SEI Technical Report number CMU/SEI-2001-TN-007, 2001.
3. L. A. Belady and M. M. Lehman; *A model of a Large Program Development*, IBM Systems Journal, 15(1):225-252, 1976.
4. K. H. Bennett and V. Rajlich; *Software Maintenance and Evolution: A Roadmap*, ICSE'2000 – Future of Software Engineering, Limerick, Ireland, pp. 73-87, 2000.
5. I. Sommerville; *Software Engineering*, Sixth Edition, Addison-Wesley, 728 p., 2001.
6. J. Ropponen and K. Lyytinen, *Components of Software Development Risk: How to Address Them? A Project Manager Survey*, IEEE Transactions on Software Engineering, 26(2), 98-112, Feb. 2000.
7. M. Ali Babar, L. Bass, I. Gorton, *Factors Influencing Industrial Practices of Software Architecture Evaluation: An Empirical Investigation*, QoSA 2007, Medford, Massachusetts, USA, July 12-13, 2007.
8. B. W. Boehm, *Software Risk management: Principles and Practices*, IEEE Software, 8(1), 32-41, January 1991.
9. M. Carr, S. Kondra, I. Monarch, F. Ulrich, and C. Walker, *Taxonomy-Based Risk Identification*, Technical Report SEI-93-TR-006, SEI, Pittsburgh, USA, 1993.
10. M. Keil, P. E. Kule, K. Lyytinen and R. C. Schmidt, *A Framework for Identifying Softare Project Risks*, Communications of the ACM, 4(11), 76-83, November 1998.
11. B. W. Boehm, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, 21(5), 61-72, May 1988.
12. A. Gemmer, *Risk Management: Moving Beyond Process*, IEEE Computer, 30(5), 33-41, May 1997.
13. H. Hecht, *Systems Reliability and Failure Prevention*, Artech House Publishers, 2004.
14. V. Clerc, P. Lago, H. van Vliet, *The Architect's Mindset*, QoSA 2007, Medford, Massachusetts, USA, July 12-13, 2007.
15. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in Software Engineering – An Introduction*, Kluwer Academic Publishers, 2002.
16. C. B. Seaman, *Qualitative Methods in Empirical Studies of Software Engineering*, IEEE Transactions on Software Engineering 25(4):557-572, July/August 1999.
17. T. C. Lethbridge, S. E. Sim, and J. Singer, Studying Software Engineers: Data Collection Techniques for Software Field Studies, Empirical Software Engineering, 10(3), 311-341, July 2005.
18. B. Kitchenham and S. L. Pfleeger, *Principles of Survey Research, Parts 1 to 6*, ACM Software Engineering Notes, 2001 – 2002.
19. D. G. Messerschmitt and C. Szyperski, *Marketplace Issues in Software Planning and Design*, *IEEE Software* **21** (3): 62–70, May/June 2004.
20. R. E. Johnson and B. Foote. "Designing Reusable Classes." Journal of Object-Oriented Programming, 1(2):22-35, 1988.

21. L. Bass, R. Nord, W. Wood, D. Zubrow, *Risk Themes Discovered Through Architecture Evaluations*, Proc. WICSA 2007
22. O'Connell, D. "Boeing's Experiences using the SEI ATAM® and QAW Processes", April, 2006, http://www.sei.cmu.edu/architecture/saturn/2006/OConnell.pdf
23. Charette, R.N., Why software fails, Spectrum, IEEE Volume 42, Issue 9, Sept. 2005 Page(s):42 – 49.

P6: Risks and Risk Management in Software Architecture Evolution: an Industrial Survey

Published in Proc. Asia Pacific Software Engineering Conference (APSEC) 2008.

Odd Petter N. Slyngstad[1], Reidar Conradi[1], M. Ali Babar[2], Viktor Clerc[3], Hans van Vliet[3]
[1]Department of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU), {oslyngst, conradi} at idi.ntnu.no
[2]LERO, University of Limerick, Limerick, Ireland, malibaba at lero.ie
[3]Department of Computer Science, Vrije Universiteit, Amsterdam, the Netherlands, {viktor, hans} at cs.vu.nl

Abstract

The little effort that has been made to study risk management in the context of software architecture and its evolution, has so far focused on output from structured evaluations. However, earlier research shows that formal, structured evaluation is not commonly used in industry. We have performed a survey among software architects, in order to capture a more complete picture of the risk and management issues in software architecture evolution. Our survey is specifically about their current knowledge of actual challenges they have anticipated and experienced, as well as strategies the respondents have employed in response. We received completely filled questionnaires from 82 respondents out of a total distribution of 511 architects from the software industry in Norway. While many of the risks we have identified can be aligned with results from earlier studies, we have also identified several risks which appear not to fit risk these risk categories. Additionally, we found a direct link to business risks, as well as a relatively low level of low levels of awareness that lack of software architecture evaluation represents a potential risk.

Keywords: *software architecture, software evolution, risk management, software architecture evaluation*

1. Introduction

Proper management of software architecture is one of the most important factors towards successful development and evolution of component-based software systems. The architecture is a core part of a software system [1], and obtaining and disseminating information about relevant risks is therefore important. By software architecture evolution, we mean accumulated changes to the structure(s) of a system of software elements, their external properties and mutual relationships [1]. Some effort has been made towards identifying and understanding the risks and corresponding strategies involved in managing software architecture evolution (but based on outputs from structured architecture evaluations – which are not commonly used in industry [7]).

We expect that practicing IT-professionals who work with architecture on a daily basis will encounter architectural evolution risks on a regular basis. Therefore, this investigation targets software professionals in the IT-industry with significant knowledge and experience about software architecture design and evolution. The systems we investigate are within Component-Based Software Engineering (CBSE – including internal, Components Off-The-Shelf (COTS) and Open Source components (OSS)) development, and whose architectures have evolved during their lifetimes. We have also targeted small and medium sized enterprises (SMEs).

Based on our initial identification of risks and management strategies [19], we have carried out an industrial survey of architectural risks and mitigation strategies in software evolution. In this investigation, we have identified additional architectural risks and associated risk management strategies related to software evolution in the IT-industry, in a three-part adapted operational matrix (Tables 2, 3, 4). This allows easy lookup of strategies and related outcome profiles as applied to the most influential risks we identified. The most influential risks were regarding poor clustering of functionality and insufficient stakeholder communication. While many of the risks we have identified fit with results from earlier studies [22][23], we have also identified several risks which appear not to fit risk these risk categories. We found a low level of awareness that lack of architecture evaluation represents a potential problem, as well as a direct link to business risks.

The remainder of this paper is organized as follows: Section 2 holds background and related work. The research method is presented in Section 3. Section 4 contains information on our data collection and analysis, and the results of our study stand in Section 5. Discussion and Threats to Validity are contained in Section 6, and conclusion and future work stand in section 7.

2. Background and Related Work

Risks are challenges that can have negative influences on a project unless they are handled properly. Risk management involves methods to handle risks that may occur during a software development project. Boehm [8] details a **risk management framework** which includes two main steps: risk assessment (i.e. risk identification, analysis and prioritization) and risk control (i.e. planning, resolution and monitoring). Technological, process and organizational issues are also important collective facets of software risk mitigation (i.e. proper handling of occurring problems to minimize their consequences) [11][12][13].

A definition of **Software Architecture** can be found in [1, page 3]: *the structure or structures of a system, which comprise software elements, the externally visible properties of those elements, and the relationships between those elements.* Defining and maintaining a software architecture properly is key to success with development and evolution of non-trivial systems, such as within CBSE [2]. Benefits to project organizational structure can also be seen. Furthermore, a lack of attention to software architecture often has negative consequences reaching beyond the architecture itself, for example with respect to inter-personal communication, unnecessary redundancies and decision making in the project [1].

Definitions of software evolution exist in the research literature in several variants: evolution as part of maintenance [5], evolution as a software lifecycle step [4], evolution as the dynamic behavior of software systems through lifelong maintenance and enhancements [3], and evolution as the enhancements and improvements regarding functionality and performance made between releases [9]. Our definition of software evolution (updated from [19]) is as follows: *the systematic and dynamic updating of a component (source code) or other artifact to a) accommodate new, altered or removed functionality or b) enhance the reliability/availability (i.e. fewer failures), performance, or other quality attribute(s) of such an artifact between different releases.*

Capturing architectural information is important in determining inherent risks in the architecture, as well as the impact these risks may have. The concept of Architectural Knowledge is relatively new in Software Engineering [21]. Combining the documented design of the architecture with records of actual design decisions, underlying assumptions and context (as well as additional factors) constitutes a knowledge base for understanding all aspects of that software architecture. The key point is that most of the needed architectural information included, save perhaps the actual architectural design, commonly is not explicitly documented [21]. Clerc et al. [14] carried out a survey among software architects in the Netherlands, focusing on architectural knowledge. Their findings show that architects emphasize creation and communication at the expense of reviewing and maintaining a given architecture. Furthermore, architects were not concerned with learning and reflection.

Ropponen and Lyytinen performed a survey regarding risks in software development and how these risks can be handled [6]. They asked project managers questions regarding software development risks, their mitigation, and corresponding influence by environmental factors. They identified the following six categories of software risk: 1) scheduling and timing risks, 2) functionality risks, 3) subcontracting risks, 4) requirements management, 5) resource usage and performance risks, and 6) personnel management risks. Their results also reveal that all of the risk categories were affected by environmental factors.

Bass et al. [22] used results from 18 ATAM evaluations to reveal and analyze risk themes specifically towards software architecture. In addition to a set of risk categories, they also found that the more prevalent risks are those of omission, i.e. of not taking action on a particular issue. They also did not find a link between the risk categories and the business/mission goals or the domain of a system. Another risk categorization from ATAM evaluations is presented in O'Connell [23], using 8 evaluation results.

These studies utilize ATAM outputs from organizations where such evaluation is established as a practice, but do not comment on how common such formal evaluation methods are in industry. A related study [7] on architecture evaluation shows that practices range from completely ad-hoc to formally structured, from qualitative to quantitative. Additional risks and management strategies could therefore escape discovery when only investigating structured analysis reports. It should also be noted that neither of these studies deal explicitly with evolution of software architecture. In contrast, our investigation incorporates risks explicitly identified in planning and experienced during the evolution of software architecture, based on input directly from software architects.

3. Research Method

Motivation: Our initial observation [19] was that risk management studies usually identify risks on the general project level [6][8][10]. On the other hand, software architecture studies commonly focus on the design, implementation and maintenance of the architecture, i.e. as a software artifact. There has been some effort to study the two in combination [19], i.e. risk management of software architecture activities, but based on outputs from structured architecture evaluations [22][23]. We therefore decided to perform a more in-depth investigation to further identify and understand actual risks and associated risk management strategies in relation to software architecture evolution, as

they are identified, experienced and employed in industry. We study industrial risk identification, analysis and prioritization (**RQ1**), as well as risk planning and resolution (**RQ2**) [8][19].

Our Research Questions (background in a review of research literature, being adapted as follows)**:**

RQ1: What are the relevant architectural evolution risks, i.e. what risks induced on the software architecture through software evolution? Boehm's first step [8] includes risk identification, analysis, and prioritization. We are focusing on the state-of-the-practice in risk awareness, i.e. we wish to gain insight regarding which risks software architects deem important with respect to an evolving architecture. In our prestudy [19] we saw that challenges (risks) were handled on a case-by-case basis, whether known before project start, or experienced during projects.

RQ2: How can these risks best be mitigated: how successful were the relevant risk management strategies? Boehm's second step [8] includes risk control, focusing on problem mitigation; i.e. proper handling of occurring problems to minimize their impact. Our goal is again to obtain an overview of state-of-the-practice. Furthermore, we wish to suggest possible improvements through enabling a systematic approach to managing architectural risks in software evolution. Hence, it is important that we obtain information regarding positive and negative aspects of applied risk management strategies and their outcomes. As mentioned, our prestudy [19] indicated that risk mitigation is performed in an ad-hoc manner. Most of the respondents also reported not having a documented or defined risk management process to deal systematically with risk aspects when altering the architecture.

Our questionnaire: Building on the questionnaire from our prestudy [19], we have designed a revised questionnaire totalling 23 questions. Questions Q1-Q5 and Q8 are closely related to those in our prestudy. Furthermore, questions Q6-Q23 were adapted and changed from an earlier empirical study aimed at identifying the factors that can influence software architecture evaluation practices [7]. Some of the questions are semi-open questions, i.e. the answer categories are indicated, but the actual answer is free text. The remaining questions are mainly closed, although some have alternatives such as "other, please describe:" to allow filling in additional answers not covered by the given alternatives for a particular question. In addition, respondent information regarding level of experience, education, number of years in the IT-industry, position, size of company etc. was collected. We have tried out the questionnaire on four of our colleagues to ensure the quality of the questions and that they were easy to understand. This caused 7 out of 23 questions to be refined.

In this article, we are focussing on identification, analysis and prioritization, as well as planning and resolution, of risks. We therefore discuss questions Q1.1 and Q1.2 of the questionnaire separately in this article, as both of them contribute to both RQ1 and RQ2. Q1.1 reads "Challenges **identified in planning the maintenance/evolution** related to the software architecture (indicate influence on the architecture, also indicate strategies, and their outcome)?", and Q1.2 reads "Challenges **experienced during the maintenance/evolution** related to the software architecture (indicate influence on the architecture, also indicate strategies, and their outcome)?"

In questions Q1.1 and Q1.2, we used a 5-point ordinal Likert scale to rank risk Influence with value range Very High (VH) = 5, High (H) = 4, Medium (M) = 3, Low (L) = 2 and Very Low (VL) = 1. Similarly, strategy Outcome was ranked Not at all = 1,

Somewhat = 2, Medium = 3, Mostly = 4, and Completely = 5 successful. The rank 0 was used to indicate "Don't know" on both scales.

Context: Our research in this investigation is on software development projects with the following two major characteristics – use of CBSE (including development with internal, COTS and OSS components), and incurred changes in the software architecture during their lifetime. This implies that the investigated projects have delivered their first production release. That is, they can be considered to be in the "post-development" phase, i.e. undergoing maintenance/evolution. The survey respondents were all from the IT-industry in Norway.

4. Data Collection and Analysis

This questionnaire-based survey was performed using a variant of snowball sampling, a technique described in [18], where key practitioners serve as contact points towards the organizations involved. The contact points are then sent the questionnaire, and forward it on to other potential respondents. The contact points can also report the total number of respondents from each organization and function as a temporary checkpoint for the number of completed questionnaires. This type of sampling is close to convenience sampling in that the contact persons are known during the execution of the survey. To ease the workload and streamline the data collection and validation process, we enabled a web-interface to make the questionnaire available to the respondents online.

To ensure previous knowledge and experience working with software architecture, we required at least 2 years of professional experience (the lowest level seen in our prestudy – respondents at this level were still able to give relevant and valuable answers). The questionnaire took about 30 minutes to fill in completely. In total we were able to reach 63 small and medium sized software companies (all less than 100 employees), with 511 potential respondents. We received 82 complete answers out of 511 total contacted (i.e. a response rate of 16 %). Furthermore, the mean project size was 7.

We analyzed the data as follows: The data on risks and strategies were divided into distinct parts and each piece coded according to risk or strategy theme(s). As an example [19], for risks this was specified as {risk condition – what may go wrong, risk consequence(s)}. For example, "requirements from earlier versions still in effect affected architecture design negatively" was coded as {earlier version requirements, negative for architecture design}. The coded pieces were then examined for commonalities and differences, combining related information pieces. For example, for risks, {earlier version requirements, negative for architecture design} and {required same functionality as before, negative for planning} were grouped as {required backward compatibility, negative for architecture maintenance/evolution planning and design}. These groups were then compared to the risks and strategies we discovered from our prestudy to check for overlaps and similarities.

We ran all the answer records through this procedure. The results were further checked to ensure reliability. This is similar to the constant comparison method described in [16]. We retained the risk classification scheme used in our prestudy [19] for the three categories technical, process and organizational. We maintain that risk

management is not merely a technical issue; rather, it covers all three categories [11][12][13].

5. Results

In presenting the results, we have divided the risks into the three categories mentioned above. The results are presented in three tables which together constitutes an adapted operational matrix. In this context they enable lookup of strategies and their outcome profiles as applied to the most influential risks we identified. The outcome is presented as a set of the number of instances each rating was given by the respondents, i.e. Outcome rating = Number of {“Not at all”, “Somewhat”, “Medium”, “Mostly”, and “Completely”} successful instances. Though it was possible to enter “Don’t Know” as a rating, none of the respondents did so. In presenting the results, the strategy with the highest number of “Completely” (or “Mostly” if no instances were rated “Completely”) successful responses is taken as the most successful. The strategy applied by the most respondents is taken as the most frequent strategy. Finally, “*” indicates that this risk was also identified among the most influential in our prestudy [19].

Technical risks: From Table 2, we can see that Technical Strategy (TS) 1 (Table 2) was the overall most successful strategy, and also the most frequent one, applied in planning. During maintenance/evolution, TS 7 (Table 2) was the most successful (and most frequent) strategy applied towards technical risks.

Table 1. Most influential (Risk Influence VH > 1) technical risks (TRs), and strategies

Technical (identified in planning), ID: Risk	**Risk Influence**	**ID:Strategy:Outcome rating =** Number of {"Not at all", "Somewhat", "Medium", "Mostly", and "Completely"} successful instances.		
TR 1: Poor clustering of functionality affected performance negatively *	VH: 7, H: 23	**TS 1**	Refactoring of the architecture	{0, 8, 2, 5, 3}
		TS 2	Redesign within constraints	{0, 0, 1, 4, 0}
		TS 3	Design with high focus on modifiability	{0, 1, 2, 6, 1}
		TS 4	Finalize modifiability design considerations early	{0, 1, 0, 0, 0}
TR 2: Requirements from other system(s) affected performance negatively	VH: 5, H: 10	**TS 2**	Redesign within constraints	{0, 1, 4, 4, 0}
		TS 5	Employ separate agents for external communication, protocol for information sharing	{0, 1, 2, 2, 0}
		TS 3	Design with high focus on modifiability	{0, 1, 4, 3, 0}
TR 3: Undefined variation points in requirements affected performance negatively, caused increased focus on modifiability	VH: 3, H: 10	**TS 3**	Design with high focus on modifiability	{0, 0, 3, 5, 1}
		TS 4	Finalize modifiability design considerations early	{0, 3, 2, 3, 0}
TR4: Extensive focus on streamlining of the architecture affected modifiability negatively	VH: 2, H: 10	**TS 3**	Design with high focus on modifiability	{0, 0, 3, 3, 1}
		TS 4	Finalize modifiability design considerations early	{0, 2, 3, 3, 0}
TR 5: Architectural mismatch caused redesign of part of the architecture	VH: 2, H: 2	**TS 1**	Refactoring of the architecture	{0, 1, 1, 0, 0}
		TS 3	Design with high focus on modifiability	{0, 0, 0, 1, 0}
		TS 4	Finalize modifiability design considerations early	{0, 0, 1, 0, 0}
Experienced during				
TR 6: Increased focus on modifiability contributed negatively towards system performance *	VH: 6, H: 10	**TS 6**	Implementation of changes towards improved modifiability	{0, 0, 2, 1, 0}
		TS 7	Minor implementation changes	{0, 1, 6, 7, 0}
TR 7: Poor original core design prolonged the duration of the maintenance/ evolution cycle *	VH: 3, H: 11	**TS 6**	Implementation of changes towards improved modifiability	{0, 0, 3, 4, 0}
		TS 8	Informal review of the architecture	{0, 0, 3, 3, 0}
		TS 7	Minor implementation changes	{0, 0, 1, 2, 0}
		TS 1	Refactoring the architecture	{0, 0, 3, 0, 0}
TR 8: Varying release cycles for COTS/OSS components made it difficult to implement required changes *	VH: 2, H: 16	**TS 9**	Use own development as potential backup solution	{0, 4, 5, 8, 0}
		TS 10	Implement extra architecture add-ons	{0, 1, 2, 0, 0}

Process risks: The results from Table 3 show that 7 out of 12 Process Strategies (PS) applied in planning were rated Completely (i.e. Outcome = 5) successful in at least one instance. For strategies applied during the maintenance/evolution, only 1 out of 12 (PS 17) strategies was rated Completely successful in one instance. Furthermore, PS 1 was the most successful and most frequent strategy used.

Table 2. Most influential (Risk Influence VH > 1) process risks (PRs), and strategies

Process (identified in planning), ID: Risk	**Risk Influence**	**ID:Strategy:Outcome rating =** Number of {"Not at all", "Somewhat", "Medium", "Mostly", and "Completely"} successful instances.		
PR 1: Lack of architecture documentation required more effort to be spent on planning during maintenance/evolution *	VH: 6, H: 25	**PS 1**	Recover needed architecture documentation using current architecture design and other artefacts as a basis	{0, 3, 2, 5, 1}
		PS 2	Thorough planning before implementing maintenance/evolution changes	{0, 1, 8, 7, 1}
		PS 3	Recover architecture evaluation artefacts where needed	{0, 0, 4, 2, 0}
		PS 4	Alter process to capture important architecture details	{0, 1, 3, 3, 0}
		PS 5	Explicit training on architecture documentation	{0, 0, 1, 3, 0}
PR 2: Lack of architecture evaluation contributed to discovering potential problems later in planning of maintenance/evolution	VH: 5 , H: 13	**PS 1**	Recover needed architecture documentation using current architecture design and other artefacts as a basis	{0, 0, 3, 4, 1}
		PS 3	Recover architecture evaluation artefacts where needed	{0, 1, 2, 4, 0}
		PS 4	Alter process to capture important architecture details	{0, 0, 2, 3, 0}
PR 3: Lack of business context analysis affected stakeholder relationships negatively	VH: 4 , H: 13	**PS 6**	Integrate business context in planning of the maintenance/evolution	{0, 2, 5, 3, 1}
		PS 7	Include business context informally	{0, 1, 1, 4, 0}
PR 4: Insufficient requirements negotiation postponed important architecture decisions	VH: 4, H: 9	**PS 8**	Negotiate requirements early	{0, 0, 2, 2, 1}
		PS 9	More explicit communication	{0, 2, 3, 0, 0}
		PS 10	Allow additional time for communication and feedback	{0, 1, 1, 3, 0}
Experienced during				
PR 5: Insufficient stakeholder communication contributed to insufficient requirements negotiation and affected implementation of new/changed architectural requirements negatively	VH: 7, H: 13	**PS 13**	Extra communication effort	{0, 1, 7, 3, 0}
		PS 14	Postpone some requirements to next maintenance/evolution cycle	{0, 0, 1, 2, 0}
		PS 15	Arrange plenary meetings for all stakeholders	{0, 0, 3, 4, 0}
		PS 16	Negotiate project extension	{0, 1, 2, 2, 0}
PR 6: Poor integration of architecture changes into implementation process affected implementation process and the architecture design negatively *	VH: 2, H: 20	**PS 17**	Overlay architecture change process onto implementation of maintenance/evolution	{0, 0, 4, 7, 1}
		PS 18	Integrate architecture considerations into implementation process	{0, 1, 9, 2, 0}

Organizational risks: Among the Organizational Strategies (OS) used in response to the most influential organizational risks (Table 4) identified in planning, 2 out of 9 (OS 2 and OS 4) were Completely successful in one instance. The highest rating for the strategies employed towards organizational risks experienced during the maintenance/ evolution was Mostly successful. Furthermore, OS 5 was the overall most frequent strategy applied towards the organizational risks identified.

Table 3. Most influential (Risk Influence VH > 1) organizational risks (ORs), and strategies

Organization (identified in planning), ID: Risk	**Risk Influence**	**ID:Strategy:Outcome rating =** Number of {"Not at all", "Somewhat", "Medium", "Mostly", and "Completely"} successful instances.		
OR 1: Not allowed to change OSS as decision mandate external to architecture team, affecting performance and modifiability negatively *	VH: 6, H: 22	**OS 1**	Frequent, interactive, scheduled meetings to keep up to date	{0, 1, 4, 5, 0}
		OS 2	Involve all "layers" of customer organization as stakeholders, allow extra time for proper communication	{0, 0, 1, 0, 0}
		OS 3	Ensure compliance with external mandate holder	{0, 0, 4, 1, 0}
		OS 4	Involve mandate holder early as stakeholder in planning	{0, 2, 4, 9, 1}
OR 2: Separate architecture team per maintenance/evolution cycle basis contributed to loss of and insufficient knowledge about the existing architectural design *	VH: 4, H: 29	**OS 5**	Dedicate personnel to "retrieve" architecture knowledge	{0, 2, 11, 6, 0}
		OS 6	Increased focus on proper documentation, to allow bringing new personnel up to speed quickly	{0, 1, 8, 6, 0}
OR 3: Cooperative maintenance/evolution with architects from customer org. required extra training and communication efforts *	VH: 3, H: 10	**OS 1**	Frequent, interactive, scheduled meetings to keep up to date	{0, 0, 1, 1, 0}
		OS 7	Perform maintenance/evolution incrementally	{0, 0, 2, 0, 0}
		OS 8	Allott extra time for proper communication with all stakeholders	{0, 0, 1, 0, 0}
		OS 9	Include other project's architects in planning, implementation	{0, 1, 4, 5 0}
OR 4: Lack of clear point of contact from customer organization contributed to inconsistencies in communication of the architecture and requirements *	VH: 2, H: 27	**OS 5**	Dedicate personnel to "retrieve" architecture knowledge	{0, 0, 1, 0, 0}
		OS 1	Frequent, interactive, scheduled meetings to keep up to date	{0, 0, 4, 5, 0}
		OS 2	Involve all "layers" of customer organization as stakeholders, allow extra time for proper communication	{0, 3, 6, 4, 1}
		OS 6	Increased focus on proper documentation, to allow bringing new personnel up to speed quickly	{0, 1, 5, 6, 0}
Experienced during				
OR 5: Prior architecture maintenance/evolution pushed to other projects due to lack of personnel influenced knowledge on the architecture negatively *	VH: 3, H: 11	**OS 10**	Regain architecture details from remaining upper management personnel	{0, 0, 2, 1, 0}
		OS 11	Keep architecture documentation centralized	{0, 0, 5, 8, 0}
OR 2: Separate architecture team per maintenance/evolution cycle contributed to loss of and insufficient knowledge about the existing architectural design *	VH: 2, H: 13	**OS 10**	Regain architecture details from remaining upper management personnel	{0, 2, 6, 6, 0}
		OS 11	Keep architecture documentation centralized	{0, 0, 0, 1, 0}
		OS 12	Set up standard procedure for distribution of architecture documentation and knowledge	{0, 2, 0, 0, 0}

Furthermore, our results show that the overall most influential risk was TR 1: "Poor clustering of functionality affected performance negatively". The corresponding most successful strategies were TS 1: "Refactoring of the architecture" and TS 3: "Design with high focus on modifiability". The second most influential risk was PR 5: "Insufficient stakeholder communication contributed to insufficient requirements negotiation and affected the implementation of new/changed architectural requirements negatively", with PS 15: "Extra communication effort" and PS 17: "Arrange plenary meetings for all stakeholders" as the corresponding most successful strategies. The outcome rating mode was "High" for Technical, "Medium","High" for Process, and "Medium" for Organizational strategies applied towards these most influential risks (Tables 2, 3, 4). The median outcome rating was "Medium" for all three categories.

6. Discussion

Comparison with related work: Table 5 shows the relation between risk categories identified by Ropponen et al. [6] (general software development risk categories) and Bass et al. [22] (architectural risk categories). The relations shown indicate the industrial relevance of the risks identified in our investigation.

Table 4. Summary of comparison to related work

Ropponen et al. [6]	**Technical risks (TR)**	**Process risks (PR)**	**Organizational risks (OR)**
Requirements risks:	TR 2, TR 3, TR 6,	PR 4	
Architecture Team risks:			OR 2
Stakeholder risks (from the subcontractor viewpoint):		PR 5	OR 3, OR 4
Bass et al. [21]			
Quality Attribute risk:	TR 6		
Integration risks:	TR 5, TR 8	PR 6	OR 1
Requirements risks:	TR 3	PR 4	
Documentation risks:		PR 1	
Process and Tools risks:		PR 2	
Allocation risks:	TR 1, TR 7, TR 8		
Coordination risks:		PR 5	OR 1, OR 2, OR 3, OR 4

In summary, three of the architectural risks we have identified do not fit the categories in related work [22][23]. We have focused specifically on architectural risks as {risk, consequence} (see Section 4), while earlier studies focused on aggregating categories of risks. Furthermore, we have identified relevant strategies towards mitigating the identified risks in software architecture evolution (Tables 2, 3, 4).

Observations on key architectural risks and promising risk management: The three-part adapted operational matrix in Tables 2, 3 and 4 enables lookup of strategies and related outcome profiles as applied to the most influential risks we identified, for both practitioners and researchers. Our aim is that researchers will use this matrix to build on in further investigations of risks and strategies in software architecture evolution. Practitioners can use this matrix to gain information on relevant strategies for use in response to risks they encounter.

Comparing our results with our prestudy [19], TR 1 (overall most influential risk) is identical to the second overall most influential risk. PR 5 (second overall most influential risk) is also closely related to the risk identified as overall most influential in that study. Furthermore, 4 out of 8 Technical risks, 2 out of 6 Process risks and 4 out of

5 Organizational risks identified in this investigation as most influential were also identified in our prestudy [19]. The larger number of identified risks appears in planning, as opposed to being encountered later during the maintenance / evolution, which was the case in our prestudy [19]. Furthermore, PR 3 shows a more **direct link to Business Risks [24]** (i.e. risks which influence software system viability) than was discovered in [19]. This comes in addition to the implicit circular feedback influences on and from e.g. normal cost and schedule monitoring.

Defined and documented architecture evaluation enables architects to e.g. discover design errors and conflicting requirements early in the process, potentially saving a project from more significant problems later. In this investigation, we find risks that mirror this concern, such as PR 2. Nevertheless, only 18 out of 82 respondents indicate this risk's influence as "Very High" or "High". While there is **a relatively low level of awareness that lack of architecture evaluation represents a potential risk**, the corresponding mitigation strategies we identified (PS 1, PS 3, PS 4) merely entail recovering the missing evaluation output. However, there is some evidence in other research that internal architecture evaluation is frequently performed by experts, and works because of their high level of competence and experience [25].

The median strategy outcome rating in all three categories (technical, process, organizational) was "Medium", indicating that there is still need for improvement in mitigating risks. While a large number of the identified Technical strategies (TS) focus on developing the specified system, the majority of the identified Process strategies (PS) involve recovering needed architectural documentation or other details. Furthermore, the majority of the Organizational strategies include efforts towards better communication with e.g. stakeholders.

The focus of architects' mitigation efforts are hence on recovering needed architecture details and improving communication while producing the system according to specification. Effort should therefore be made towards improving regular documentation and evaluation of the architecture, integrated with the maintenance / evolution process. Proper training of both architects and organizational management are means to achieve these improvements.

Threats to Validity: We here discuss validity threats in our investigation, based on Wohlin et al. [15] (specifically for experiments, but also applicable to survey studies):

Construct Validity: The research questions have a firm basis in the research literature. The actual questionnaire questions have been mapped directly to the research questions. The survey questionnaire has been further pre-tested through four colleagues to ensure its quality. The questionnaire questions that were not adopted from a previous study [7] (see Section 3) were initially investigated in our prestudy [19]. Furthermore, all terminology used in the questionnaire is explained at the start of the questionnaire to provide clear definitions and avoid misinterpretations.

External Validity: This survey has been conducted by using non-probabilistic snowball-like sampling [18]. It is very difficult to achieve a random sample in surveys within the software engineering field, due to the lack of good demographic information regarding the populations we are interested in, though an example of stratified-random sampling of projects has been described in the research literature [20]. Furthermore, we ensured the total sampling frame (511 professionals) had relevant background and experience in software architecture. All the respondents are nevertheless from the Norwegian IT-industry, an issue which remains a limitation.

Internal Validity: All the respondents had relevant knowledge of and experience with industrial software development. They have also expressed an interest in the survey, so we think that they have answered the survey questions to the best of their ability by relying on their own experiences, skills and knowledge of software architecture. We were also available via email during the survey to clarify any ambiguities in the questions or the accompanying definitions, in addition to the provided terminology definitions in the questionnaire.

Conclusion Validity: This is a qualitative study, and we have used non-probabilistic snowball-like sampling. The number of respondents is 82 (out of 511), and were all from small and medium companies (all less than 100 employees), with a mean project size of 7 person-years.

7. Conclusion and Future Work

Our survey on risks and risk management regarding software architecture evolution has involved 82 respondents from the software industry. Through this survey on state-of-practice, we have identified real, industrial, architectural risks and corresponding management strategies employed in response.

We have developed a three-part adapted operational matrix (Tables 2, 3, 4) for risks and corresponding risk management strategies in software architecture evolution, based on responses from our survey respondents. Table 6 shows a summary of our findings.

Table 5. Summary of findings

Most influential risks	**Corresponding most successful strategies**
1. "Poor clustering of functionality affected performance negatively"	"Refactoring of the architecture" and "Design with high focus on modifiability"
2. "Insufficient stakeholder communication contributed to insufficient requirements negotiation and affected the implementation of new/changed architectural requirements negatively"	"Extra communication effort" and "Arrange plenary meetings for all stakeholders"
Additional findings: • Direct link to Business risks. • Relatively low level of awareness that lack of architecture evaluation represents a potential risk.	

Future work involves expanding our research on risk and risk management issues to include other countries (e.g. Netherlands) in our survey base. Furthermore, we want to couple these risks and corresponding risk management strategies with an investigation of code-level and artefact data related to architecture evolution, in order to move towards a framework for better handling of these issues. Finally, a more thorough analysis of data pertaining to software architecture evaluation methods, processes and issues is planned.

8. Acknowledgements

This research is performed in cooperation between NTNU, Vrije Universiteit Amsterdam, and the Lero center in Limerick. We thank all parties involved. The study was performed in the SEVO (Software EVOlution in component-based software engineering) project (SEVO, 2007), a Norwegian R&D project in 2004-2008 with contract number 159916/V30. It also falls under the Griffin project umbrella at Vrije Universiteit Amsterdam.

9. References

[1] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Second Edition, Addison-Wesley, 2004.

[2] L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, K. Wallnau, *Volume I: Market Assessment of Component-based Software Engineering* in SEI Technical Report number CMU/SEI-2001-TN-007, 2001.

[3] L. A. Belady and M. M. Lehman; *A model of a Large Program Development*, IBM Systems Journal, 15(1):225-252, 1976.

[4] K. H. Bennett and V. Rajlich; *Software Maintenance and Evolution: A Roadmap*, ICSE'2000 – Future of Software Engineering, Limerick, Ireland, pp. 73-87, 2000.

[5] I. Sommerville; *Software Engineering*, Seventh Edition, Addison-Wesley, 728 p., 2004.

[6] J. Ropponen and K. Lyytinen, *Components of Software Development Risk: How to Address Them? A Project Manager Survey*, IEEE Trans. Sw. Engr., 26(2):98-112, Feb. 2000.

[7] M. Ali Babar, L. Bass, I. Gorton, *Factors Influencing Industrial Practices of Software Architecture Evaluation: An Empirical Investigation*, Proceedings of QoSA 2007, Springer LNCS 4880, pp. 90-109, Medford, Massachusetts, USA, July 12-13, 2007.

[8] B. W. Boehm, *Software Risk management: Principles and Practices*, IEEE Software, 8(1), 32-41, January 1991.

[9] P. Mohagheghi and R. Conradi, *An Empirical Study of Software Change: Origin, Acceptance Rate, and Functionality vs. Quality Attributes*, ISESE 2004, Redondo Beach (Los Angeles), USA, 19-20 Aug. 2004.

[10] M. Keil, P. E. Kule, K. Lyytinen and R. C. Schmidt, *A Framework for Identifying Softare Project Risks*, Communications of the ACM, 4(11), 76-83, November 1998.

[11] B. W. Boehm, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, 21(5), 61-72, May 1988.

[12] A. Gemmer, *Risk Management: Moving Beyond Process*, IEEE Computer, 30(5), 33-41, May 1997.

[13] H. Hecht, *Systems Reliability and Failure Prevention*, Artech House Publishers, 2004.

[14] V. Clerc, P. Lago, H. van Vliet, *The Architect's Mindset*, Proceedings of QoSA 2007, Springer LNCS 4880, pp. 231-249, Medford, Massachusetts, USA, July 12-13, 2007.

[15] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in Software Engineering – An Introduction*, Kluwer Academic Publishers, 2002.

[16] A. L. Strauss and J. M. Corbin, *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*, Sage Inc., 1998.

[17] T. C. Lethbridge, S. E. Sim, and J. Singer, *Studying Software Engineers: Data Collection Techniques for Software Field Studies*, Empirical Software Engineering, 10(3):311-341, July 2005.

[18] B. Kitchenham and S. L. Pfleeger, *Principles of Survey Research, Parts 1-6*, ACM Software Engineering Notes, 2001–2002.

[19] O. P. N. Slyngstad, J. Li, R. Conradi, M. Ali Babar, *Identifying and Understanding Architectural Risks in Software Evolution: An Empirical Study*, Accepted at Profes '08, 15p, forthcoming in a Springer LNCS, June 2008.

[20] R. Conradi, J. Li, O. P. N. Slyngstad, V. B. Kampenes, C. Bunse, M. Morisio and M. Torchiano, *Reflections on conducting an international survey of Software Engineering*, in J. Verner and G. H. Travassos (Eds.): Proc. Int'l Symposium on Empirical Software Engineering (ISESE'05), pp. 214-223, Noosa Heads (Brisbane), Australia, 17-18 Nov. 2005, IEEE Computer Society, 2006.

[21] P. Kruchten, P. Lago, H. van Vliet, Timo Wolf, *Building up and Exploiting Architectural Knowledge*, Proc. WICSA 2005, pp. 291-292, IEEE Computer Society 2006.

[22] L. Bass, R. Nord, W. Wood, D. Zubrow, *Risk Themes Discovered Through Architecture Evaluations*, Proc. WICSA 2007, pp. 1-10, IEEE Computer Society, 2007.

[23] D. O'Connell, *Boeing's Experiences using the SEI ATAM® and QAW Processes*, April, 2006, http://www.sei.cmu.edu/architecture/saturn/2006/OConnell.pdf

[24] D. G. Messerschmitt and C. Szyperski, *Marketplace Issues in Software Planning and Design*, *IEEE Software* **21** (3): 62–70, May/June 2004.

[25] C. Hofmeister, P. Kruchten, R. L. Nord, H. Obbink, A. Ran, P. America, Generalizing a Model of Software Architecture Design from Five Industrial Approaches, Proc. WICSA 2005, pp. 77-88, IEEE Computer Society 2006.

Appendix B

In this appendix, the abstracts of the 10 secondary articles that contribute towards the background of the work in this thesis are presented. They are presented in the same order as earlier in this thesis. The articles are titled as follows:

SP1: Preliminary Results from a State-of-Practice Survey on Risk Management in Off-The-Shelf Component-Based Development.

SP2: Barriers to Disseminating Off-The-Shelf Based Development Theories to IT Industry.

SP3: An Empirical Study on Off-the-Shelf Component Usage in Industrial Projects.

SP4: Validation of New Theses on OTS-Based Development.

SP5: Reflections on conducting an international survey of Software Engineering.

SP6: An Empirical Study on the Decision Making Process in Off-The-Shelf Component Based Development.

SP7: A State-of-the-Practice Survey of Off-the-Shelf Component-Based Development Processes.

SP8: An Empirical Study of Software Changes in Statoil ASA – Origin, Priority Level and Relation to Component Size.

SP9: A Case Study of Defect-Density and Change-Density and their Progress over Time.

SP10: A State-of-the-Practice Survey on Risk Management in Development with Off-The-Shelf Software Components.

SP1: Preliminary Results from a State-of-the-Practice Survey on Risk Management in Off-The-Shelf Component-Based Development

Jingyue Li[1], Reidar Conradi[1,2], Odd Petter N. Slyngstad[1], Marco Torchiano[3] , Maurizio Morisio[3], and Christian Bunse[4]

[1] Department of Computer and Information Science,
Norwegian University of Science and Technology (NTNU),
NO-7491 Trondheim, Norway
{jingyue, conradi, oslyngst}@idi.ntnu.no

[2] Simula Research Laboratory, P.O.BOX 134, NO-1325 Lysaker, Norway

[3] Dip.Automatica e Informatica, Politecnico di Torino
Corso Duca degli Abruzzi, 24, I-10129 Torino, Italy
{morisio, marco.torchiano}@polito.it

[4] Fraunhofer IESE, Sauerwiesen 6,
D- 67661 Kaiserslautern, Germany
Christian.Bunse@iese.fraunhofer.de

Abstract. Software components, both Commercial-Off-The-Shelf and Open Source, are being increasingly used in software development. Previous studies have identified typical risks and related risk management strategies for what we will call OTS-based (Off-the-Shelf) development. However, there are few ef-fective and well-proven guidelines to help project managers to identify and manage these risks. We are performing an international state-of-the-practice survey in three countries - Norway, Italy, and Germany - to investigate the rela-tive frequency of typical risks, and the effect of the corresponding risk man-agement methods. Preliminary results show that risks concerning changing re-quirements and effort estimation are the most frequent risks. Risks concerning traditional quality attributes such as reliability and security of OTS component seem less frequent. Incremental testing and strict quality evaluation have been used to manage the possible negative impact of poor component quality. Real-istic effort estimation on OTS quality evaluation helped to mitigate the possible effort estimation biases in OTS component selection and integration.

SP2: Barriers to Disseminating Off-The-Shelf Based Development Theories to IT Industry

Jingyue Li[1], Reidar Conradi[1,2], Odd Petter N. Slyngstad[1], Christian Bunse[3], Umair Khan[3], Maurizio Morisio[4], Marco Torchiano[4]

[1]Dept. of Computer and Info. Sci. Norwegian Univ. of Sci. and Tech.
NO-7491 Trondheim, Norway
{jingyue, conradi, oslyngst}
@idi.ntnu.no
[2]Simula Research Laboratory, P.O.BOX 134, NO-1325 Lysaker, Norway
[3]Fraunhofer IESE, Sauerwiesen 6,
D-67661 Kaiserslautern, Germany
{Christian.Bunse, khan}@iese.
fraunhofer.de
[4]Dip. Automatica e Informatica, Politecnico di Torino
Corso Duca degli Abruzzi, 24,
I-10129 Torino, Italy
{maurizio.morisio, marco.torchiano}@polito.it

ABSTRACT

In this position paper, we have reported results of an industrial seminar. The seminar was intended to show our findings in an international survey, conducted in Norway, Italy and Germany, on off-the-shelf component-based development. Discussion in the second section of the seminar revealed several obstacles of popularizing the OTS based development theories into IT industry.

SP3: An Empirical Study on Off-the-Shelf Component Usage in Industrial Projects

Jingyue Li[1], Reidar Conradi[1,2], Odd Petter N. Slyngstad[1], Christian Bunse[3], Umair Khan[3], Marco Torchiano[4], and Maurizio Morisio[4]

[1]Department of Computer and Information Science,
Norwegian University of Science and Technology (NTNU),
NO-7491 Trondheim, Norway
{jingyue, conradi, oslyngst}@idi.ntnu.no

[2]Simula Research Laboratory, P.O.BOX 134, NO-1325 Lysaker, Norway

[3]Fraunhofer IESE, Sauerwiesen 6,
D- 67661 Kaiserslautern, Germany
{Christian.Bunse, khan}@iese.fraunhofer.de

[4]Dip. Automatica e Informatica, Politecnico di Torino
Corso Duca degli Abruzzi, 24, I-10129 Torino, Italy
{maurizio.morisio, marco.torchiano}@polito.it

Abstract. Using OTS (Off-The-Shelf) components in software projects has become increasing popular in the IT industry. After project managers opt for OTS components, they can decide to use COTS (Commercial-Off-The-Shelf) components or OSS (Open Source Software) components instead of building these themselves. This paper describes an empirical study on why project decisionmakers use COTS components instead of OSS components, or vice versa. The study was performed in form of an international survey on motivation and risks of using OTS components, conducted in Norway, Italy and Germany. We have currently gathered data on 71 projects using only COTS components and 39 projects using only OSS components, and 5 using both COTS and OSS components. Results show that both COTS and OSS components were used in small, medium and large software houses and IT consulting companies. The overall software system also covers several application domains. Both COTS and OSS were expected to contribute to shorter time-to-market, less development effort and the application of newest technology. However, COTS users believe that COTS component should have good quality, technical support, and will follow the market trend. OSS users care more about the free ownership and openness of the source code. Projects using COTS components had more difficulties in estimating selection effort, following customer requirement changes, and controlling the component's negative effect on system security. On the other hand, OSS user had more difficulties in getting the support reputation of OSS component providers.

SP4: Validation of New Theses on OTS-Based Development

Jingyue Li[1], Reidar Conradi[1,2], Odd Petter N. Slyngstad[1], Christian Bunse[3], Umair Khan[3], Marco Torchiano[4] and Maurizio Morisio[4]

[1]*Department of Computer and Information Science,*
Norwegian University of Science and Technology (NTNU),
NO-7491 Trondheim, Norway
{jingyue, conradi, oslyngst}@idi.ntnu.no

[2]*Simula Research Laboratory, P.O.BOX 134, NO-1325 Lysaker, Norway*

[3] *Fraunhofer IESE, Sauerwiesen 6,*
D- 67661 Kaiserslautern, Germany
{Christian.Bunse, khan}@iese.fraunhofer.de

[4]*Dip. Automatica e Informatica, Politecnico di Torino*
Corso Duca degli Abruzzi, 24, I-10129 Torino, Italy
{maurizio.morisio, marco.torchiano}@polito.it

Abstract

Using OTS (Off-The-Shelf) components in software development has become increasingly popular in the IT industry. OTS components can be either COTS (Commercial-Off-The-Shelf), or OSS (Open-Source-Software) components. A recent study with seven structured interviews concluded with six theses, which contradicted widely accepted (or simply undisputed) insight. Since the sample size of that study was very small, it is necessary to investigate these theses in a larger and randomized sample. A state-of-the-practice survey in three countries – Norway, Italy, and Germany – has been performed to validate these new theses. Data from 133 OTS component-based projects has been collected. Results of this survey support four and contradict two of the initial theses. The supported theses are: OSS components were mainly used without modification in practice; custom code mainly provided additional functionality; formal OTS selection processes were seldom used; OTS component users managed to get required changes from vendors. The unsupported theses are: standard mismatches were more frequent than architecture mismatches; OTS components were mainly selected based on architecture compliance instead of function completeness.

SP5: Reflections on conducting an international survey of Software Engineering

Reidar Conradi 1), Jingyue Li 1), Odd Petter N. Slyngstad 1), Vigdis By Kampenes 2), Christian Bunse 3), Maurizio Morisio 4), Marco Torchiano 4),

1) Dept. of Computer and Information Science, Norwegian University of Science and Technology, NO-7491 Trondheim, Norway, {conradi, jingyue, oslyngst} at idi.ntnu.no

2) Simula Research Lab, P. O. Box 134, NO-1325 Lysaker, Norway, vigdis at simula.no

3) Fraunhofer IESE, Sauerwiesen 6, D-67661 Kaiserslautern, Germany, bunse at iese.fraunhofer.de

4) Dip. Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129 Torino, Italy, {morisio, torchiano} at polito.it

Abstract

Component-based software engineering (CBSE) with Commercial Off-The-Shelf (COTS) or Open Source (OSS) Components are more and more frequently being used in industrial development. We therefore need to issue experience-based guidelines for the evaluation, selection and integration of such components. We have performed a survey on industrial COTS/OSS development in three countries – Norway, Italy and Germany. Concrete survey results, e.g. on risk management policies and process tailoring, are not being described here, but in other papers. This is a method paper, reporting on the challenges, approaches and experiences gained by conducting the main survey. The main contributions are as follows: At best, we can achieve a stratified-random sample of ICT companies, followed by a convenience sample of relevant projects. This is probably the first software engineering survey using census type data, and has revealed that the entire sampling and contact process can be unexpectedly expensive. It is also hard to avoid national variances in the total process, possibly leading to uncontrollable biases.

SP6: An Empirical Study on the Decision Making Process in Off-The-Shelf Component Based Development

Jingyue Li[1], Reidar Conradi[1,2], Odd Petter N. Slyngstad[1]
[1]Department of Computer and Information Science, Norwegian University of Science and Technology (NTNU), NO-7491 Trondheim, Norway
[2]Simula Research Laboratory, P.O.BOX 134, NO-1325 Lysaker, Norway
{jingyue, conradi, oslyngst}@idi.ntnu.no

Christian Bunse[3]
[3]Fraunhofer IESE, Fraunhoferplatz 1, D-67663 Kaiserslautern, Germany Christian.Bunse@ iese.fraunhofer.de

Marco Torchiano4, Maurizio Morisio[4]
[4]Dip. Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi, 24, I-10129 Torino, Italy {marco.torchiano,maurizio.morisio}@polito.it

ABSTRACT

Component-based software development (CBSD) is becoming more and more important since it promotes reuse to higher levels of abstraction. As a consequence, many components are available being either open-source software (OSS) or commercial-off-the-shelf (COTS). However, it is still unclear how the decision for acquiring OSS or COTS components is made in practice. This paper describes an empirical study on why project decision makers selected COTS instead of OSS components, or vice versa. The study was performed as an international survey in Norway, Italy and Germany. It focused on decision making on using off-the-shelf (OTS) components. We have gathered answers from 83 projects using only COTS components and 44 projects using only OSS components. Results of this study show significant differences and commonalities of integrating OSS or COTS components. Moreover, the study illustrates several research questions that warrant future research.

SP7: A State-of-the-Practice Survey of Off-the-Shelf Component-Based Development Processes

Jingyue Li[1], Marco Torchiano[2], Reidar Conradi[1],

Odd Petter N. Slyngstad[1], and Christian Bunse[3]

1 Department of Computer and Information Science,
Norwegian University of Science and Technology (NTNU),
NO-7491 Trondheim, Norway
{jingyue, conradi, oslyngst}@idi.ntnu.no
[2] Dip. Automatica e Informatica, Politecnico di Torino
Corso Duca degli Abruzzi, 24, I-10129 Torino, Italy
marco.torchiano@polito.it
[3] Fraunhofer IESE, Fraunhoferplatz 1,
D-67663 Kaiserslautern, Germany
Christian.Bunse@iese.fraunhofer.de

Abstract. To gain competitive advantages software organizations are forced to develop systems quickly and cost-efficiently. Reusing components from third-party providers is one key technology to reach these goals. These components, also known as OTS (Off-the-Shelf) components, come in two different types: COTS (Commercial-Off-The-Shelf) and OSS (Open–Source-Software) components. However, the reuse of pre-fabricated components bears one major question: How to adapt development processes/methods with refer to system development using OTS components. To examine the state-of-the-practice in OTS component-based development a survey on 133 software projects in Norway, Italy and Germany was performed. The results show that OTS-based development processes are typically variations of well-known process models, such as the waterfall- or prototyping model, mixed with OTS-specific activities. One reason might be that often the process is selected before the use of OTS components is considered. Furthermore, the survey shows that the selection of OTS components is based on two processes: "Familiarity-based" and "Internet search-based". Moreover, it appears that the lifecycle phase to select OTS components is significantly correlated with a project members' previous familiarity with possible OTS candidates. Within this paper, we characterize the state-of-the-practice concerning OTS processes, using seven scenarios, and dis-cuss how to decide or modify such processes and how to select OTS components.

SP8: An Empirical Study of Software Changes in Statoil ASA – Origin, Priority Level and Relation to Component Size

Anita Gupta, Odd Petter N. Slyngstad, Reidar Conradi, Parastoo Mohagheghi
Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway

Harald Rønneberg, Einar Landre
Statoil KTJ/IT
Stavanger (Forus), Norway

Abstract—This paper describes the results of analyzing change requests from 4 releases of a set of reusable components developed by a large Oil and Gas company in Norway, Statoil ASA. These components are total 20348 SLOC (Source Lines of Code), and have been programmed in Java. Change requests in our study cover 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 are 208 combined perfective, adaptive and preventive changes. The results reveal that 59% of changes are perfective, 27% of changes are adaptive and 14% of changes are preventive. The corrective changes (223 in total) are excluded in this paper, since they will be analyzed in future work. We have also investigated the relation between customers' and developers' priority on change requests and found no significant difference between customer and developers' priority of change requests. Larger components had more change requests as expected and priority level of change requests increases with component size. The results are important in that they characterize and explain the changes to components. This is an indication as to which components require more effort and resources in managing software changes at Statoil ASA.

SP9: A Case Study of Defect-Density and Change-Density and their Progress over Time

Anita Gupta, Odd Petter N. Slyngstad, Reidar Conradi, Parastoo Mohagheghi
Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
{anitaash, oslyngst, conradi, parastoo} at idi.ntnu.no

Harald Rønneberg, Einar Landre
Statoil KTJ/IT
Forus, Stavanger
{haro, einla} at statoil.com

Abstract

We have performed an empirical case study, investigating defect-density and change-density of a reusable framework compared with one application reusing it over time at a large Oil and Gas company in Norway, Statoil 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 analyzed 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-density both for the reusable JEF framework and the application. The results revealed that the defect-density of the reusable framework was lower than the application. The JEF framework had higher change-density in the first release, but lower change-density than the DCF application over the successive releases. For the DCF application, on the other hand, a slow increase in change-density appeared. On the relation between change-density and defect-density for the JEF framework, we found a decreasing defect-density and change-density. The DCF application here showed a decreasing defect-density, with an increasing change-density. The results show that the quality of the reusable framework improves and it becomes more stable over several releases, which is important for reliability of the framework and assigning resources.

SP10: A State-of-the-Practice Survey on Risk Management in Development with Off-The-Shelf Software Components

Jingyue Li, Member, IEEE Computer Society, Reidar Conradi, Member, IEEE,
Odd Petter N. Slyngstad, Student Member, IEEE, Marco Torchiano, Member, IEEE Computer Society, Maurizio Morisio, Member, IEEE Computer Society, and Christian Bunse

Abstract—An international survey on risk management in software development with Off-the-Shelf (OTS) components is reported upon and discussed. The survey investigated actual risk-management activities and their correlations with the occurrences of typical risks in OTS component-based development. Data from 133 software projects in Norway, Italy, and Germany were collected using a stratified random sample of IT companies. The results show that OTS components normally do not contribute negatively to the quality of the software system as a whole, as is commonly expected. However, issues such as the underestimation of integration effort and inefficient debugging remain problematic and require further investigation. The results also illustrate several promising effective risk reduction activities, e.g., putting more effort into learning relevant OTS components, integrating unfamiliar components first, thoroughly evaluating the quality of candidate OTS components, and regularly monitoring the support capability of OTS providers. Five hypotheses are proposed regarding these risk-reduction activities. The results also indicate that several other factors, such as project, cultural, and human-social factors, have to be investigated to thoroughly deal with the possible risks of OTS-based projects.