An Empirical Study of Software Quality and Evolution in the Context of Software Reuse

*TDT4735 Software Engineering, Depth Study*

*20th December 2005*

Mari Torgersrud Haug
Thea Christine Steen

NTNU
Norwegian University of Science and Technology, NTNU
Department of Computer and Information Science, IDI
Abstract

Software reuse has become increasingly popular among development projects in the software industry. It promises decreased time-to-market, reduced development costs and increased software quality. Another subject that has received more attention over the last decades, is software evolution. Software evolution is a consequence of the inevitably maintenance of software performed to ensure high levels of software quality. Unfortunately, software evolution does not always affect software quality in a purely positive way. There is a need for more empirical studies on the software quality and evolution in the context of reuse. This report describes the results of a contribution to this subject. We have defined 3 groups of hypotheses regarding the evolution of defect-density and stability in reusable components over several releases. Our results indicate that defect-density in the studied components decrease as time passes and new versions are released. The results also indicated that there is a relationship between the stability, defined as change-density, and defect-density. Our tests showed that the two variables are linearly dependent. The linearly dependency between change-density and defect-density imply that components with high change-density also have high defect-density. Industrial software development projects in general, and the team at Statoil ASA that developed the studied components especially, ought to consider our results. Furthermore, our results can be used to make a prediction model for new releases of the studied components. We have also evaluated the defect reporting schemes and change request schemes used by the development team, and made recommendations for improvements. The design of these schemes can have great impact on the team’s software process and product.
Preface

This report is part of the course TDT4735 Software Engineering at the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU). The motive of this course is to give a more extensive understanding of software engineering through a self-imposed project assignment from within the special field.

The report is a part of a study of software development that utilize reusable components at Statoil ASA. The Ph.D. students Anita Ashok Gupta and Odd Petter Nord Slyngstad, will perform further studies on the basis of our results. Gypta and Slyngstad participate in the project Software EVOlution in Component-Based Software Engineering (SEVO) at NTNU.

We wish to thank our project advisors that made it possible for us to achieve our goals in this project. First, we want to thank our project supervisor, Professor Reidar Conradi, for providing insightful input and valuable feedback throughout the project. Second, we want to thank the Ph.D. students Anita Ashok Gupta and Odd Petter Nord Slyngstad, for providing us with data from Statoil, and giving us advice and support regarding our research and report writing. We would like to thank the SEVO project for inviting us to their meetings and providing us with insight into their work, as well as advice and feedback valuable to our work. Furthermore, we want to thank the developers at the system development department at Statoil ASA at Rotvoll, Trondheim, for giving us access to their project data. We also thank Professor Tor Stålhane for his advice regarding statistical analysis.

Trondheim, December 20th 2005.

Thea Christine Steen
Mari Torgersrud Haug
# Contents

1 Introduction ............................... 1
   1.1 Research motive ........................ 1
   1.2 Report outline .......................... 2

2 State of the Art ............................ 3
   2.1 Software engineering .................... 3
      2.1.1 An introduction to software engineering ................ 3
      2.1.2 Software engineering in the 1960’s and today .............. 4
   2.2 Software systems ........................ 5
   2.3 Software engineering characteristics and challenges ............ 5
   2.4 Software reuse .......................... 6
      2.4.1 Reuse dimensions ...................... 6
      2.4.2 Reuse of intermediate products ............... 7
      2.4.3 Perspectives of software reuse and reuse within application domains ....................... 8
      2.4.4 Non-technical aspects of software reuse ........ 8
   2.5 Component Based Software Engineering(CBSE) .................... 9
      2.5.1 COTS and OSS .......................... 10
   2.6 Software evolution ........................ 12
      2.6.1 Simple staged and version staged software development .... 16
   2.7 Software Quality .......................... 19
      2.7.1 Taxonomies on quality attributes ............... 19
      2.7.2 Software defects ....................... 19
   2.8 Related work ............................. 20
      2.8.1 Studies of Fault-Density .................. 20
      2.8.2 Studies of fault-density in the context of reuse ....... 21
   2.9 An empirical study of defect reports ........................ 24
      2.9.1 Work related to software evolution ................ 25
   2.10 Research in Software Engineering ..................... 25
      2.10.1 Empirical Strategies .......................... 26
      2.10.2 Measurement ............................. 26
### CONTENTS

2.10.3 Variables .................................................. 27  
2.10.4 Data analysis .............................................. 28  
2.10.5 Validity of results ........................................ 30  

3 State of the practice at Statoil .................................. 31  
3.1 Statoil context .................................................. 31  
  3.1.1 Statoil Java Enterprise Framework (SJEF) .................. 31  
  3.1.2 Java Enterprise Framework (JEF) .......................... 32  
  3.1.3 The SJEF projects ........................................ 32  
3.2 Data from various projects ..................................... 34  
  3.2.1 Test stages ................................................. 34  
  3.2.2 Test planning ............................................... 34  
  3.2.3 Test reports ............................................... 36  
  3.2.4 Trouble reporting .......................................... 36  
  3.2.5 Classification of changes .................................. 36  
  3.2.6 ClearCase .................................................. 38  
  3.2.7 ClearQuest ................................................ 38  

4 Research agenda .................................................. 41  
4.1 Introduction .................................................... 41  
  4.1.1 Overall research goal ..................................... 41  
  4.1.2 Terminology ............................................... 41  
4.2 Formulation of research questions .............................. 42  
  4.2.1 Initial set of research questions ......................... 43  
  4.2.2 Research questions anno October 26th 2005 ............... 45  
  4.2.3 Final set of research questions ......................... 46  
4.3 Research Strategy ............................................. 48  

5 Results ............................................................ 49  
5.1 Raw data ....................................................... 49  
5.2 Data analysis ................................................... 49  
  5.2.1 Defect-density ............................................ 50  
  5.2.2 Stability .................................................. 51  
  5.2.3 Stability vs. Defect-density ............................... 53  
5.3 Discussion and Summary of the results ......................... 56  
5.4 Validity of results ............................................ 57  
  5.4.1 Conclusion validity ....................................... 57  
  5.4.2 Internal validity .......................................... 58  
  5.4.3 Construct validity ........................................ 58  
  5.4.4 External validity ......................................... 58  
5.5 Evaluation of the change reporting in Statoil ................ 58
List of Figures

2.1 The Simple Staged Model ........................................ 17
2.2 The Version Staged Model ........................................ 18

3.1 Overview of the Java Enterprise Framework .................... 32
3.2 Enhancement request workflow .................................... 37
3.3 Defect workflow .................................................. 38

5.1 Relation between change-density and defect-density .......... 54
5.2 Residual plot between change-density and defect-density .... 55
List of Tables

2.1 Lehman’s Laws .............................................. 13
2.2 ISO 9126 ......................................................... 20
2.3 Threats, attributes and means to/of dependability .......... 21
2.4 Research hypotheses and results ............................ 23
2.5 Research questions implied by P. Mohageghi in her study of Ericsson 23
2.6 Lessons learned regarding metrics and data collection .... 25
2.7 ANOVA table .................................................... 29

3.1 Components in JEF ............................................ 33
3.2 Attributes in change reports ................................. 39

4.1 Initial set of research questions .............................. 43
4.2 Metrics ............................................................ 44
4.3 Set of research questions anno October 26th .............. 46
4.4 Set of research questions anno 22th November 2005 .... 46

5.1 Size of JEF components, in #LOC .......................... 50
5.2 Defect-density in JEF components, #DR/KLOC .......... 50
5.3 Average defect-density in each release .................... 51
5.4 ANOVA table for testing H1 ................................. 51
5.5 Changes per component size(KLOC) in JEF components 52
5.6 Average change-density data ................................ 52
5.7 ANOVA table for testing H2 ................................. 52
5.8 Average change-density and defect-density per component 53
5.9 Correlation table ............................................... 54
5.10 Summary of the regression results ......................... 55
5.11 Summary of the results from the hypothesis testing .... 56
Chapter 1

Introduction

This chapter presents the projects research motive and the report outline.

1.1 Research motive

Software engineering has matured to become an engineering field with great impact on today’s society [1]. Software developers develop systems for the government, all kinds of application domains in industry and different businesses. The field of software engineering is exposed to different challenges. Some of these are delayed project deliveries, exceeded budgets and poor software quality [2] [1]. In order to handle these challenges, various emerging phenomena are studied and new methodologies and techniques are developed, studied and evaluated.

Software reuse has become increasingly popular in the development of software products because it promises reduced time-to-market, less development effort and increased software quality [3] [4]. However, there are also risks introduced by software reuse, such as unknown quality properties, economic instability of the COTS component vendor [5] and difficulties in following customer requirement changes and COTS component updates [4].

Another subject within the field of software engineering, that has recently been widely recognized and accepted as worthy of serious study is software evolution [6]. This phenomenon is a necessary consequence of the inevitable software maintenance process. Software maintenance is performed to correct errors (corrective maintenance), to adapt the system to changes in the outside world (adaptive maintenance) and satisfactorily support new requirements as they arise (perfective maintenance) [6] [7]. The resulting evolution of the software will affect the products internal complexity, and as time passes, make it increasingly harder to reason about and maintain the system [6]. Therefore, software evolution may cause a decline in software quality with ageing.

In cooperation with Anita A. Gypta and Odd-Petter N. Slyngstad, and under the supervision of Reidar Conradi, we have defined and tested 3 groups of hypotheses regarding the software quality and evolution of reusable components. We will try to verify these hypotheses using data from two projects at Statoil ASA. These projects develop two application systems on the basis of reusable components based on the Java Enterprise Framework. Our goal is to draw some conclusions regarding the quality and evolution of reusable software components in these projects. We hope that our results will be useful for future work related to the studied topics in general, and to Gypta and Slyngstad’s further studies, specifically.
1.2 Report outline

The report is structured into the following chapters:

**Chapter 1 - Introduction:** This chapter presents the project and the setting of the project. It also gives an outline of the report.

**Chapter 2 - State of the Art:** A summary of the literature survey will be given in this section. The focus lie on software reuse, software evolution and software quality. The chapter also present other studies that our project got motivation and proposals of possible research goals. At least we will describe how empirical studies are performed in the context of software engineering.

**Chapter 3 - State of the practice at Statoil:** This chapter describe our research context at Statoil ASA with the SJEF architecture. We are going to look at components implements to accomplish the SJEF architecture, and how the defect and change report process are taking place.

**Chapter 4 - Research Agenda:** This chapter presents our project goal, the coarse and formulation of research questions with basis in relevant literature and our resource strategy.

**Chapter 5 - Results:** Our own contribution is reported in this chapter. We have formulated the research questions as hypothesis, found dataset to test this hypothesis. We will present the results from this tests and discuss the meaning.

**Chapter 6 - Conclusion and Further work:** This chapter conclude with what we have discovered in our research. There will also be a section that presents how our work can be taken further.
Chapter 2

State of the Art

This chapter presents the literature study that was done prior to our case study. It gives an introduction to the field of software engineering with emphasis on software reuse, software evolution and software quality.

2.1 Software engineering

This section presents an introduction to software engineering and its challenges. Then, it presents software reuse and an introduction to OTS-components and how it is utilized to ease challenges met by software developers. Further, the phenomenon software evolution is introduced and described on the basis of among other Lehman’s Law. This is followed by an introduction of software quality that focus upon faults and stability as quality attributes. The chapter is finished by a summing up performed work that is related to our study.

2.1.1 An introduction to software engineering

To clarify what is meant by software engineering we present the definition stated by The IEEE Standard Glossary of Software Engineering Terminology.

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.


Software engineering emerged from the field of computer science, which cause the to fields to have an considerable overlap [1]. However, computer science is more influenced by mathematics and physics. Software engineering, on the other hand, has to deal with the management of development projects, human factors, and cost estimation and control, in addition to the engineering of software [1].

Li et al. claim that software is a generic term for computer programs including systems programs which operate the computer itself, and application programs which control the particular task at hand [8]. The definition of a software system given by Li et al. is as follows.

A software system is the final product delivered by the actual project,
consisting of application plus components, but again excluding platform software.

- Li et al. [8]

This definition is based on the following definitions of a software component and a software application also given by Li et al. [8].

Software components are program units of independent production, acquisition, and deployment that can be composed into a functioning system.

- Li et al. [8]

A software application is code (built in-house) to provide certain functionalities of the system.

- Li et al. [8]

Software systems are built to make a model of the real world and solve a problem. To construct a software system the problem is analyzed and requirements are identified and precisely described. The design choices are made to satisfy the requirements given by the users and customers. There are different process that capture the phases that a software development project should go through. The basic phases are requirements engineering, design, implementation, testing and maintenance.

2.1.2 Software engineering in the 1960’s and today

This subsection and subsection 2.3 is loosely based on [1].

The field of software engineering sprung out of computer science in the 1960’s. This was a very young research field that had its origin in the mathematics. At this time, programming was still considered an art and programmers had little or no formal education [1]. They learned by doing. Furthermore, the programs they built were often quit small and written by only one person. The users were experts in the application area concerned or the scientists them selves. The problems they encountered were mostly of a technical or mathematical nature, and they were attempted to be solved by adding more and more programmers to the project. This, so-called million-monkey approach, often lead to what is known as the software crisis [1]. This is a state were the product is delivered to late, did not behave as the user expected, were rarely adapted according to changed circumstances and error wasn’t detected until after the software had been delivered to the customer.

Today’s software systems are of a quite different nature than these early-day’s programs. Their development time is often spread over months or even years, and a software development project involves a lot of different kinds of people all working on the same system during different stages of its development [1]. This indicates how both the size and the complexity of systems developed today are considerably greater than in the early days. Furthermore, the present-day end users are not only experts, but all kinds of people. These end users come from all kinds of application domains, and many of these domains are unfamiliar to the software developers. This lack of expert knowledge among the engineers is a very peculiar feature of software engineering and development [1].
CHAPTER 2. STATE OF THE ART

2.2 Software systems

There are a number of kinds of software systems. Some are developed to act like they are intelligent (Artificial Intelligence), others are developed for private and personal usage, and even others support business and industry processes. These are just a sample of the different kinds of software system that exist in today’s society. Information systems systems has received more attention over the last two decades. These systems constitute a central tool in many business processes and across different application domains. The Internet-based reference book, Wikipedia, gives the following definition of an information system.

A system, whether automated or manual, that comprises people, machines, and/or methods organized to collect, process, transmit, and disseminate data that represent user information.

- Wikipedia [9]

Many organizations benefited from the introduction an information system in their organization to support their business processes and strategy. An information system require a storage unity, units supporting different business tasks and applications on top supporting different users.

Some organizations have various information systems in different sections of their business. These systems will constitute a total business system of different information systems that might have difficulties communicating with each other and give consistent data. Enterprise systems are introduced as a solution to this problem. An Enterprise Information System (EIS), is a comprehensive software package that incorporates all modules needed to run the operations of a business [10]. Accordingly, the organization needs only one single EIS. This will handle all of it business processes and sections. The benefits of this systems is that all the parts of the information system can communicate with each other and be more efficient than a number of stand alone systems distributed in the various organization sections [10].

2.3 Software engineering characteristics and challenges

Software engineering has a lot in common with other engineering practices [1]. In both the construction of software and the construction of bridges and buildings, we work from a set of desired functions and use scientific and engineering techniques in a creative way. The product is developed in a number of phases, these phases are planned carefully, the development process is audited continuously, the construction is based on a clear and complete design and so on [1].

However, software engineering also has its very own facets that separates it from traditional engineering practices [1]. Since copying software is almost free the cost of constructing software is incurred with development and not during production. Furthermore, software doesn’t ware out in the way that bridges and buildings do. Software maintenance may actually cause the software system to wear out. Other challenges related to software engineering are visibility of progress, continuity in the development process and the impact that small changes have on the product [1]. Imprecise measurements of progress and the customary underestimation of the total effort needed for the project cause problems to accumulate as time elapses. Furthermore, changes in the software specification may cause considerable changes in the software itself and a great amount of re-work with associated delays and added costs. On the other hand, small changes in the software itself may have considerable effects on the software functionality and quality [1].
20th December 2005

CHAPTER 2. STATE OF THE ART

The list below summarize the challenges of software engineering recognized by Hans van Vliet [1].

- Software engineering concerns the construction of large programs.
- The central theme is mastering complexity.
- Software evolves.
- The efficiency with which software is developed is of crucial importance.
- Regular cooperation between people is an integral part of programming-in-the-large.
- The software has to support its users effectively.
- Software engineering is a field in which members of one culture create artifacts on behalf of members of another culture.

- Hans van Vliet [1]

2.4 Software reuse

This section gives an introduction to the concept of software reuse. This is done by presenting the way software reuse may differ along different dimensions, the reuse of intermediate products, perspectives on software reuse and reuse within application domains, and a presentation of the non-technical aspects of software reuse.

2.4.1 Reuse dimensions

Software reuse may differ along different dimensions. These dimensions are substance, scope, approach, technique, usage and product. In the following each of these dimensions and their characteristics are described [11].

The dimension substance describes the essence of the things that are reused. The element may be a procedure on e.g. how to carry out inspection or prototyping, or a product. Example product elements are a component in some programming language, a generic algorithm, or a concept [11].

The scope of the software reuse may be horizontal or vertical. Generic components as e.g. mathematical subroutines and GUI widgets, that can be used across multiple application domains have a horizontal scope. A component that can only be used within one application domain has a vertical scope [11].

There are also different approaches to software reuse. The approach may be planned or opportunistic [11]. In the planned approach, development for reuse, the developers plan for and work actively with building reusable components before the system that uses these are built. This approach demands considerable investments and effort up front. The software should be developed with reuse in mind, and there must be placed an extra emphasis on software testing and documentation [11]. When the opportunistic approach, development with reuse, is used, components are reused only when and if the developers happen to know of these, and if they fit. The components are not developed with reuse in mind [11]. Therefore, there are no extra up-front investments or effort associated with this approach to reuse. However, the selection procedure and adaptation of the component to be reused and the software that is going to use it, to one another, might be quite time and cost consuming [11].
The composition-based and the generation-based technique are two different software reuse techniques. The composition-based technique reuses components by composing a new system from existing components [11]. The components are passive fragments that are retrieved from an existing base [11]. The generation-based technique, on the other hand, reuses knowledge by placing it in a program that generates another program. The element of reuse is a pattern that is used to actively generate the target system [11].

The element of reuse may be used as a black-box or white-box. The black-box usage use the element as-is [11]. That is, the elements inside, such as source code or procedure description, is not read or modified. Only the elements interface is visible. The white-box usage may read and modify the element before they are incorporated [11].

The actual objects of reuse, the product, may be source code, design, architecture, an object, text, and so on [11].

### 2.4.2 Reuse of intermediate products

Complex software can be difficult to understand by others than its developers. Therefore, software documentation is needed. Documentation of the systems requirements, design, implementation, and testing is often used to make the software easier to understand and maintain. Furthermore, such documentation and enhanced maintainability may also lead to reuse of the software and a more controlled software evolution, as argued in section 2.6. However, reuse of intermediate product, such as frameworks and middleware, can make it even easier to understand the software and facilitate maintainability and controlled software evolution, as well as further reuse of the software. Frameworks and middleware is defined by D.C. Schmidt and F. Buschmann as quoted.

*Frameworks provide both a reusable product line architecture for a family of related applications and an integrated set of collaborating components that implement concrete realizations of the architecture.*

- D.C. Schmidt, F. Buschmann [12]

*Middleware is reusable software that leverages patterns and frameworks to bridge the gap between functional requirements of applications and the underlying operating systems, network protocol stacks and databases.*

- D.C. Schmidt, F. Buschmann [12]

Over the past few years a number of middleware standards have emerged, for example Java 2 Enterprise Edition (J2EE). Research show that frameworks can enable effective development and reuse of middleware. As a result, middleware is now commonly developed using frameworks and they guide the integration so that the middleware meet the functional as well as the quality requirements for the application domain more effectively than developing from scratch [12].

As mentioned above, many different kinds of products are objects for reused; source code, design, architecture, components, text, and procedures. Architectures, designs and frameworks are examples of intermediate products [11]. Reuse of design or application framework can make the system’s composition and functionality realization easier to grasp. The structure and much of the code can be recognized
and easily understood by developers familiar with these reused intermediate products \[11\]. Furthermore, this makes it easier to maintain and modify the software correctly without harming it \[11\].

The architecture found in compilers is an example of a reused architecture. Most compilers are composed in the same way with a lexical analyzer, parser, symbol table, code generator \[11\]. This architecture has for a long time been more or less the only architecture that was heavily reused. Today reusable product line architectures and application frameworks are emerging for different application domains. These are called Domain Specific Software Applications (DSSA) \[11\].

Other kinds of reused intermediate products are skeletons or templates \[11\]. These products are reused by instantiation of unfinished components and filling in holes in the code. It is important to memorize that there is a certain trade-off between the number of holes that needs to be filled in, in relation to the component’s size, and the profit of reusing the product \[11\].

### 2.4.3 Perspectives of software reuse and reuse within application domains

Different software projects develop software that fall to various application domains. Tailored-made is software that is developed for one single customer and usage specifically. Such software is more critical for some application domains than others. For domains that need tailor-made software it is suitable to do a domain analysis, which identifies, captures, structures and organizes the information for reuse. This analysis will reassure that the right domain knowledge and the needed functionality and quality are captured with the right kind of abstraction in the component \[11\].

Furthermore, it is important that the component reflects the primitive notations of the application domain. To achieve this, one should consider requirements for similar systems in the same application domain in addition to the requirements of the system being developed at the moment \[11\]. This approach is suitable for the development of reusable software components, and can result in the creation of a product line architecture or an application framework.

### 2.4.4 Non-technical aspects of software reuse

Hans van Vliet recognize the factors given in the list below as essential for the success of software reuse \[11\].

- Unconditional and extensive management support,
- establishment of an organizational support structure,
- incremental program implementation,
- significant success, both financial and organizational,
- compulsory or high incentives,
- domain analysis was conducted either consciously or unconsciously, and
- explicit attention to architectural issues, such as common architecture across a product line.

- Hans van Vliet \[11\]

Furthermore, van Vliet identifies the following costs associated with software reuse \[11\].
• Initial development costs of the component to be reused,
• the direct and indirect costs of including the components in a library, and
• the costs of (possibly) adapting the component and incorporating it into the system under development.

It is very important to memorize that the development of reusable components is only profitable when these components indeed are reused. When the component is actually reused suitably, the developers are saved a lot of work [13]. This is verified by several empirical studies. Therefore, software reuse is called capital-intensive work [11]. The initial costs of this kind of work are high, but it returns over a longer period. Labor-intensive work, on the other hand, is associated with months of work, succeeded by a reaping of the profit as soon as the project is finished [11]. Development without reusable software is associated with the latter kind of work.

The management plays an essential role in the success of software development projects [11]. In order for these projects to succeed, they must receive attention and sufficient resources. Unfortunately, the management is often too busy with assuring that the current projects is finished within time and budget to think about the success of future projects [11]. Furthermore, there is no real incentive to pursue reusability. It is just another tool used to meet the time and resource constraints of the project. These may the main reasons why software reusability tends to have low priority [11].

The management’s foci and attitude has such a major impact on project success because the management is the institution that delegates the resources to the project and because its attitude affects the project workers and software developers greatly [11]. If the management doesn’t pay attention to longer-term goals, there is no reason why the software developers should. The developers must be given the right incentives, and the management must pay attention to the developers expectations about the nature of their work [11]. Furthermore, reuse of components seems to be increased if the components embody abstractions the programmers are familiar with. Therefore, the management must stress that reusable components are developed in a procedure that foci well-known standards and interfaces, and good documentation [11].

2.5 Component Based Software Engineering(CBSE)

This section presents the component based software engineering, and COTS and OSS, as well as the definitions, characteristics and usage, motivations and risk associated with COTS and OSS.

Component Based Software Engineering (CBSE) is an approach to software development based on extensive use of existing components.

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.


CBSE is based on the assumption that sufficient commonality exist among large software systems to justify developing reusable components to exploit and satisfy commonality. A definition of CBSE follows below:
Component-based software engineering is concerned with the rapid assembly of systems from components that satisfy the following conditions.

- The components and frameworks have certified properties, and
- these certified properties provide the basis for predicting the properties of systems built from components.

F. Bachmann et al. [15]

A.W. Brown and K.C. Wallnau write that the entry of development environments, with supporting tools for the programmers and languages like C++ and Java, made it possible to share and distribute applications [14]. This approaches provide communication and coordination necessary to decompose applications into pieces. Another implication for the evolution of CBSE is the requirement from organizations. Organizations began to understand how their software-intensive systems could support their business processes and strategy, which can give advantages over competitors [14]. Software technology has became the core part of business enterprises in all market sectors. As a consequence the customers demand more flexible enterprise systems that can evolve as needed to meet changing business environment and needs [14]. It is preferable not to develop everything from scratch when a new application is required. CBSE give opportunity to reuse core functionality across applications and encapsulate organizational best practices. To meet the need for change and fast development of new functionality, CBSE enable flexible upgrade and replacement of system pieces [14]. The pieces can be components developed in house, by third parties or off the shelf. This will be further described in the next section 2.5.1.

2.5.1 COTS and OSS

The term reusable or Of-The-Shelf (OTS) components is used as a generic term which covers a great amount of different kinds of products. Some are developed in the context of a specific project and then recognized as reusable components, while others are intended to be reusable from the very start of their development. These components may either be developed for commercial purposes, be meant for a bank of reusable components in the company that built them, or they may be shared in an open source system community. The latter kind of reusable products have an open source code and are shared freely. Accordingly, they are called Open Source Systems (OSS). Products developed with commercial intentions are named Commercial Of-The-Shelf (COTS) components.

Definitions

The research field of COTS covers a wide range of components. It spans from simple algorithms and user interfaces, through complex operating systems and middleware, to tool packages and complex application packages [10]. Additionally, the vendor might extend the COTS product in customer-specific ways during a COTS tender. Examples of this kind of extensions are integration with other systems or addition of so-called glueware. Consequently, the COTS term covers many different kinds of products and component. This, and the fact the research field is rather new, immature and informal, causes a lack of a common terminology and uncertainties and disagreements regarding the absolute definition of COTS components. While some communities define COTS to be only those products which are owned commercially by a vendor and have a closed source code [17], others argue that OSS should be considered as a sort of COTS even if it is free and has an open source code. The
reason for this argumentation is that these researchers mean that the openness of OSS code is not fully utilized [18]. In the contrary, it just acts as a guarantee in case of technical problems, they proceed.

In our further work, we will recognize COTS as products which are commercially owned by a vendor and has a closed source code. OSS, on the other hand, will be regarded as systems with open source code, which are freely available through OSS communities.

Characteristics

The COTS users are more concerned with whether the software component follows the trends in the marketplace than the OSS users [13]. Often the customer tends to influence vendor on product evolution. Consequently the COTS components must be accommodated according to these trends, in addition to the continuous trade-offs made between the customers requirements and the system’s software architecture. Other factors that customers mention as critical in the selection process are the components architecture, its compliance with standards, and whether the customer is familiar with the product or its generic architecture [18]. The fact that COTS’ source code is closed makes it more difficult for the customer to examine the quality of the software component and ties the customer to the vendor for maintenance support [5]. Consequently, the COTS users or customers are willing to pay the vendor to ensure software quality and they expect that the vendor to will give good support [13].

OSS’ open source code gives the users the opportunity to examine the software code themselves and make the necessary adjustment. This brings us to the idea behind open source:

When programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, and fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing.

- www.opensource.com [19]

Furthermore, OSS is often developed without software tasks such as documentation, testing and field support. However, OSS users seem to be most comfortable with investigating and adjust or improve the source code themselves [13]. Even if OSS are spread in such a freely manner, they often have a license holder, who should be consulted before integrating the system in to a product.

Usage, motivation and risks

This subsection is based on the article An Empirical Study on Off-the-Shelf Component Usage in Industrial Projects by Li et al. Both COTS and OSS are used in small, medium and large software and IT consulting companies, and in association with several different application domains [13]. The users of both COTS and OSS are motivated by shorter time-to-market, less development effort, and higher reliability [13]. The COTS users are more concerned with how well the product follows trends in the marketplace than the OSS users. COTS users also trust the COTS quality and COTS vendor support ability [13]. The key motivation among OSS users is possibly free source code. These users enjoy the possibility to access the source code directly themselves rather than contacting the products vendor [13].
COTS users seem to have difficulties with following requirement and COTS evolution. OSS users, on the other hand, manage this challenge much better, but are less satisfied with user support from the OSS providers [13].

2.6 Software evolution

This section presents software evolution on the basis of studies and work performed by, among others, M. Lehman, K.H. Bennett and V.T. Rajlich.

As large software systems are developed over a period of several years, their structure tends to degrade and it becomes more difficult to understand and change them. Difficult changes are excessively costly or require an excessively long interval to complete.

- T.L. Graves et al. [20]

The phenomenon described by Graves et al. in the quotation above is known as program or software evolution. Meir Lehman was the first to become aware of this phenomenon. This happened when he was working on Project IMP for IBM in late 1968 [21]. He realized that the real problem lay not in the design of a specific system, but in the methods and methodology of design. Software and system designs were unlikely ever to be optimum or complete. System application and system potential evolve [21]. Lehman’s observations at IBM went largely unrecognized at the time [21], but Lehman’s interest for the topic had been wakened and he kept on doing research on it. Today he is one of the most recognized researchers in the special field of software evolution. One of his contributions is the Lehman Law’s given in table 2.1.

Lehman distinguish between two types of systems, E-type system sytems and S-type system systems. An E-type is a systems that is used and embedded in a real-world domain [22]. Once such systems are operational, they are judged by the results they deliver [6]. A S-type system can be verified, i.e. proven correct [22]. A correct program, or system, possesses all the properties required to satisfy its specification [6].

The term evolution is used in various ways by different researchers and communities. Some use it exclusively for the events after the initial implementations. Others expand the term’s scope to also encompass the initial development of the system, in addition to its subsequent maintenance [23].

Although studied for over thirty years, the phenomenon of software evolution and its economic and social importance have only recently been widely recognized and accepted as worthy of serious study [6]. The increasing awareness of the phenomenon is, in part, due to computers’ pervasiveness, their growing employment in domains such as industry, commerce, government, and the increasing utilization of the Internet. The growing organizational dependence on software systems has resulted in widespread recognition of a need for continuous and effective co-evolution of business and software [6]. Furthermore, massive and unexpected software evolution during development and maintenance of software threatens the quality of the software and the delivery of products and services on time and budget. User expectations, technologies, personnel, companies are among the factors that are in a constant flux [23]. In addition, as users become more sophisticated and dependent on satisfactory system operation, their need for speedy, reliable and cost-effective evolution of their software systems increases [6].
CHAPTER 2. STATE OF THE ART

Table 2.1: Lehman’s Laws

<table>
<thead>
<tr>
<th>No., year</th>
<th>Brief Name</th>
<th>Law</th>
</tr>
</thead>
<tbody>
<tr>
<td>I, 1974</td>
<td>Continuing change</td>
<td>E-type systems must be continually adapted else they become progressively less satisfactory in use.</td>
</tr>
<tr>
<td>II, 1974</td>
<td>Increasing complexity</td>
<td>As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it.</td>
</tr>
<tr>
<td>III, 1974</td>
<td>Self regulation</td>
<td>Global E-type system evolution processes are self-regulating.</td>
</tr>
<tr>
<td>IV, 1978</td>
<td>Conservation of Organisational Stability</td>
<td>Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving E-type system tends to remain constant over product lifetime.</td>
</tr>
<tr>
<td>V, 1978</td>
<td>Conservation of Familiarity</td>
<td>In general, the incremental growth and long term growth rate of E-type systems tend to decline.</td>
</tr>
<tr>
<td>VI, 1991</td>
<td>Continuing Growth</td>
<td>The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime.</td>
</tr>
<tr>
<td>VII, 1996</td>
<td>Declining Quality</td>
<td>Unless rigorously adapted to take into account changes in the operational environment, the quality of E-type systems will appear to be declining.</td>
</tr>
</tbody>
</table>

Due to the need of co-evolution of software and business, and the constant flux in user expectations, technologies, personnel and companies, the original system requirements may change to reflect changing business, user and customer needs and the software system may need to be changed accordingly [6] [24] [25]. Other changes occurring in a software system’s environment may emerge from undiscovered errors during system validation, challenging repair activities or when new hardware is introduced [26]. M. Postema et al. describe the following as the possible changes to a software system [25].

- Simple changes to correct errors in source code.
- Extensive changes to correct errors in the software design.
- Significant changes to correct specification errors or incorporate new requirements.

Accordingly, software evolution is strongly related to software maintenance [27] [25] [28]. Software maintenance is defined by [7] as:

*The process of modifying a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.*
Maintenance is usually segmented into different classifications. The definitions for the different types of maintenance vary amongst practitioners in the field [26]. Corrective, adaptive, preventive and perfective maintenance are the most frequently used classifications. The IEEE Std. 610 12-1990 give the following definitions of these [7].

**Corrective maintenance**: Maintenance performed to correct faults in hardware or software.

**Adaptive maintenance**: Software maintenance performed to make a computer program usable in a changed environment.

**Perfective maintenance**: Software maintenance performed to improve the performance, maintainability, or other attributes of a computer program.

**Preventive maintenance**: Maintenance performed for the purpose of preventing problems before they occur.

The costs of these maintenance operations are additionally higher than that of developing the original software product. Maintenance can be defined as the single most expensive activity in software engineering, requiring 65% to 75% of total effort [25] [29]. A study conducted by Lientz and Swanson in the late 1970’s showed that around 75% of the maintenance effort was of the adaptive and perfective kind, while error correction consumed about 21% [28]. Many subsequent studies suggest a similar magnitude of the problem. These studies show that the incorporation of new user requirements is the core problem for software evolution and maintenance [28]. Many changes required are actually those that the original designers can’t even conceive of. Hence, maintenance is inevitable. This is recognized by Bennett and Rajlich. They emphasize the following as reasons for the importance of software maintenance [28].

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

They state that these are enduring problems because of the continuing evolution of technology and needs of the business makes [28]. Furthermore, they suggest that the following topics should be emphasized and studied further [28].

- Architectures which allow considerable anticipated change in the software without comprising system integrity and invariants.
- Architectures which in themselves can evolve in controlled ways.
- Managing the knowledge and expertise of the software team.

Bennett and Rajlich’s claim is supported by Lehman and his laws given in table 2.1. As time passes, user experience increases, user needs and expectations change, new opportunities arise, system application expands in terms of number of users or details of usage and so on [6] [24]. Furthermore, there will be changes in the real world that will impact the application system or it’s domain of operation. This imply changes to the system to ensure that it restores its characteristic of being
an acceptable model of the domain \[6\]. These facts are reflected by Lehman’s First Law, and lead to its practical outcome in an unending need for software maintenance \[6\]. That is, a continued evolution of the product.

Lehman’s Second Law states that as the system evolves, and with the number of elements \(n\), the work required to ensure a correct and adequate interface between the new and the old, the potential for error and omission, the likelihood of the incompatibility between assumptions, all tend to increase as \(n^2\). Moreover, the inter-connectivity increases as time passes because the changes and additions that are made are likely to be ever more remote from the initial design concepts and architecture. Accordingly, the complexity of the system increases \[6\]. This growth in system complexity and its causes, tend to be accompanied by a decline in the software quality and in the evolution rate of the product. Therefore, additional work must be undertaken to compensate for this increase in system complexity and decline in software quality and evolution rate\[6\]. The guidelines provided by Lehman on how to handle challenges, are a release-based, or version staged, development approach, that alternate between releases focusing on quality improving (anti-regressive) and progressive \[30\] activities \[6\].

Lehman’s Third Law says that global E-type system evolution processes are self-regulating. Lehman reports that a common feature among observed growth patterns of release-based systems is a superimposed ripple. He believes that these ripples reflect the action of stabilizing mechanisms that yield the regulation referred to by this law. As many other business processes, software development has outer loop mechanisms. The goal of outer loop mechanisms is stability, that is planned and controlled change, in technical, financial, organisational and sales terms \[6\]. To achieve this goal the various processes (technical, business, marketing, management, organizational, support and user related) and the humans who work in, influence or control them, apply positive (reinforcing) and negative (constraining) information controls. Their actions, such as progress and quality monitoring and control, checks, balances and control on resource usage, represent feedback based mechanisms as described by the third law \[6\].

To handle such the feedback system described by the Lehman’s Third Law, identification of the feedback mechanisms and controls that participate in self-stabilization, followed by exploitation of these in future planning, management process improvement, is needed \[6\]. Lehman suggests the application of measurement and modeling techniques and establishment of baselines that are continually improved as new data becomes available as valuable steps in order to identify of the feedback mechanisms and controls. After these are identified, the observations and founds should be used in the planning, management and process improvement associated with future software development \[6\].

Lehman’s Eight Law is related to the Third Law and state that the E-type evolution processes is a feedback system. Lehman states that this implies that feedback constraints the ways the evolution process’ constituents interact with one another and will modify their individual, local, and collective, global behavior. For sound process planning, management and improvement, the feedback nature of the process must be mastered \[6\]. As described above, the software process is vulnerable to positive and negative feedback. For, for example, the growth and stabilization processes described by Lehman’s First and Third Law, positive feedback conveys the desire for functional extension leading to pressure for growth and a need for continuing adaptation to exogenous changes. Excessive pressure may result in too extensive changes in the software. This can lead to instability and, furthermore, the final break up of the system. To create stability in the process, the management should create negative feedback in response to information on progress, system qual-
ity and so on. This negative feedback should take form as directives and controls
to limit change, contain side effects and drive the process in the desired direction[6]. This will result in a stabilization of the feedback system, which is profitable for
the software evolution[6].

2.6.1 Simple staged and version staged software development

Lehman suggest a release-based development approach. Bennett and Rajlich had
emphasized the difference between the evolution of the software product in the
release-based development approach in contrast to the regular, simple development
approach[28]. He defines these two approaches to software development as version
staged and simple staged, respectively.

The simple staged model in figure 2.1 illustrates the software development lifecycle
[28] associated with the regular software development approach where the product
is developed to satisfy its total set of requirements and has one single delivery date.
The model’s key contribution is the separation of the maintenance phase into an
evolution stage followed by a servicing and phase-out[28]. Bennett and Rajlich states
that the goal of the evolution stage is to adapt the application to the ever-changing
user requirements and operating environment to utilize the software’s success in the
marketplace[28]. This is consistent with the descriptions given in section 2.6. When
the software stops evolving, it enters the servicing stage, where only small tactical
changes are possible. In the phase-out stage, the software is still in production, and
at last, in the close-down stage, the software use is disconnected and the users are
directed towards a replacement[28].

The versioned staged model in figure 2.2 is an amplification of the simple staged
model that illustrates the development lifecycle for a software development project
that produces versions, or releases, of the software product during an extended
evolution phase[28]. Each version, or release, is a working system that satisfies a
subset of the total system’s requirements[31]. The subset of requirements realized
by a release, or version, of the product may overlap with, and it is often an expansion
of, subsets realized by earlier releases. In the case of versioned software development,
all substantial changes in the functionality are implemented in future versions[28].

The version staged model shown in figure 2.2 is illustration of what is called an incre-
mental development lifecycle. Such development methodologies are often adopted
to face the challenge of changing requirements and environments[31], stated in section2.6. Furthermore, this kind of development approach introduce the benefits of
earlier detection of misunderstandings or inconsistencies among the requirements,
designs and implementation. It also introduces the benefit of continuous objective
verification of the software quality and the users’ satisfaction, since each version has
a significantly shorter lead time[32]. Incremental development also encourage con-
tinuous learning and improvement of the software process and product through the
whole development process[2]. This is very rewarding and useful in association with
the process of identification and exploitation of feedback mechanisms and controls
to stabilize the feedback system the software development and evolution process is
a part of.

However, release-based development introduces new questions regarding the assess-
ment and prediction of quality and maintainability or evolution of the software
product across releases. Research question recognized by P. Mohahgehghi are given
in the list below[31].

- How can we compare quality of releases or components in different releases?
Figure 2.1: The Simple Staged Model
Figure 2.2: The Version Staged Model
• Can results of assessments of one release be used for prediction of future releases?
• How long is a release maintained before it is closed down?
• How much are components modified between releases?

2.7 Software Quality

Software quality is defined by IEEE. 1061-1992 as the degree to which software possesses a desired combination of attributes [33]. Furthermore, [33] specifies that this combination of attributes shall be clearly defined; otherwise the assessment of software quality is left to intuition. For the purpose of this standard, defining software quality is equivalent to defining a list of software quality attributes required for that system. These software quality attributes should be measured by a set of appropriate software metrics. This will reduce subjectivity by providing a quantitative basis for decision-making on software quality and make software quality more visible [33].

2.7.1 Taxonomies on quality attributes

The first elaborate studies on the notation of software quality appeared in the late 1970’s [34]. Since then different taxonomies on software quality attributes have been presented. Among the first are McCall’s taxonomy from 1977 [34]. This taxonomy presents two levels of quality attributes; those that can be objectively or subjectively measured and those that are external attributes and only can be indirectly measured. McCall name these quality criterions and quality factors, respectively. Quality factors are given by the degree of which one or more quality criterions are satisfied by the software [34].

A recent ISO standard, ISO 9126, has been made to define a set of quality characteristics [34]. The scheme of this standard is hierarchical to avoid the interrelation between characteristics. In this standard each sub-characteristic is related to exactly one quality characteristic. Furthermore, ISO 9126 only speak of a software product. Ergo, it does not concern itself with the issues of process quality. The standard stand out by concentrating on the user view on quality, whereas other taxonomies often adress the product view on the same [34]. Table 2.2 presents the ISO 9126 quality characteristics and sub-characteristics.

2.7.2 Software defects

J. Laprie et al. defines fault, error, and failure as threats to the quality attribute dependability [35]. This quality attribute is not found in the ISO 9126 standard, but discussed by among others [36], [35], [37] and [38]. There are multiple definitions of the quality attribute and disagreements on what its sub-attributes are. J. Laprie et al. defines it as the ability to deliver service that can justifiably be trusted, and devides the concept into three parts: threats, attributes and means. These are given with their respective sub-attributes in table 2.3. Because these are threats to the quality attribute dependability, they are threats to the software quality of the product at hand as well.

A fault is defined by [7] as an incorrect step, process or data definition. It is the adjudged or hypothesized cause of an error, and is passive until it produces an error [35]. An error is an activated fault [39] and the part of a system that causes failure [7]. The error propagates a failure when the delivered service deviates from
Table 2.2: ISO 9126

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Sub-characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality</td>
<td>Suitability</td>
</tr>
<tr>
<td></td>
<td>Accuracy</td>
</tr>
<tr>
<td></td>
<td>Interoperability</td>
</tr>
<tr>
<td></td>
<td>Security</td>
</tr>
<tr>
<td>Reliability</td>
<td>Maturity</td>
</tr>
<tr>
<td></td>
<td>Fault tolerance</td>
</tr>
<tr>
<td></td>
<td>Recoverability</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Time behavior</td>
</tr>
<tr>
<td></td>
<td>Resource utilization</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Analyzability</td>
</tr>
<tr>
<td></td>
<td>Changeability</td>
</tr>
<tr>
<td></td>
<td>Stability</td>
</tr>
<tr>
<td></td>
<td>Testability</td>
</tr>
<tr>
<td>Portability</td>
<td>Adaptability</td>
</tr>
<tr>
<td></td>
<td>Installability</td>
</tr>
<tr>
<td></td>
<td>Co-existence</td>
</tr>
<tr>
<td></td>
<td>Replaceability</td>
</tr>
</tbody>
</table>

the correct service [39]. Thus, a failure is visualized by the transition from correct to incorrect service [35]. Error and failures are often referred to as active threats [39]. Furthermore, the origin of faults are classified as either human or machine [40].

Software stability

Software stability is a quality attribute related to software evolution, as described in section [2.6]. The software process’ outer-loop mechanisms provide the process with positive information that pressure for growth and adaptation as well as negative information that limits change, contains side effects and drives the process in the desired direction [6]. If the information provided by the feedback loops and control mechanisms of the software process is handled correctly, the result is stability. Stability, in terms of software, can be described as controlled software evolution, and avoidance of the growth in software complexity, that result in a decline of software quality, with aging.

2.8 Related work

This section presents related work that our study is inspired by. The four subsections present studies of fault-density, fault-density in the context of reuse, and defect reporting, as well as work related to software evolution.

2.8.1 Studies of Fault-Density

Ostrand and Weyuker [41], Fenton et al. [42] and Malaiya and Denton [43] are among those who have studied the relation between fault-density and parameters such as software size, complexity, requirement volatility, software change history, or software development practices. Their results on whether or not there is a relation
Table 2.3: Threats, attributes and means to/of dependability

<table>
<thead>
<tr>
<th>Part</th>
<th>Sub-attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threats</td>
<td>Faults</td>
</tr>
<tr>
<td></td>
<td>Errors</td>
</tr>
<tr>
<td></td>
<td>Failures</td>
</tr>
<tr>
<td>Attributes</td>
<td>Availability</td>
</tr>
<tr>
<td></td>
<td>Reliability</td>
</tr>
<tr>
<td></td>
<td>Safety</td>
</tr>
<tr>
<td></td>
<td>Confidentiality</td>
</tr>
<tr>
<td></td>
<td>Integrity</td>
</tr>
<tr>
<td></td>
<td>Maintainability</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Fault prevention</td>
</tr>
<tr>
<td></td>
<td>Fault tolerance</td>
</tr>
<tr>
<td></td>
<td>Fault removal</td>
</tr>
<tr>
<td></td>
<td>Fault forecasting</td>
</tr>
</tbody>
</table>

between defect-density and component size, and how this relation degenerates, are contradictory.

Ostrand and Weyuker studied faults of 13 releases of an inventory tracking system at AT&T [41]. Their results showed that fault-density slowly decreases with component size, and that files including high number of faults in one release, remain fault-prone in later releases. They also observed that new files had higher fault-density than older files.

Fenton et al. specifically studied the relation between the number of faults and module size in the context of release-based development. They didn’t find any relation between the fault-density and module size. However, they found that size weakly correlates with the number of pre-release faults, but don’t correlate with post-release faults [42].

Another interesting study, is one performed by Malaiya and Denton. This study analyzed several studies and introduced the theory of an optimal module size that follows the scale of economy [43]. This theory is based on two mechanism; the partitioning of software into modules and how the modules are implemented. They found that along the first dimension, faults decline as module size grows. Along the second dimension, the implementation of module, the faults increase with the module size. The right combination of values along these to dimensions, gives an optimal module size.

2.8.2 Studies of fault-density in the context of reuse

The studies presented above don’t study fault-density in the context of software reuse. Just a few empirical studies actually study fault-density in this context [40]. This section presents some studies that do.

Graves et al. [20] studied the change history of 80 modules of a legacy system developed in C, and some application-specific languages, to build a prediction model for future faults. Their observations indicated that such a prediction model should be based on the change history of the modules, regarding the number of changes, length of changes, time elapsed since the changes were made. Component size and complexity metrics were not proper for such prediction. Furthermore, they concluded that recent changes contributed to most of the fault potential.
Melo, Briand and Basili performed a study to assess the impact of reuse in software quality and productivity in object-oriented systems [44] [45]. The population that was studied was the graduate level class offered by the Department of Computer Science at the University of Maryland. The students were randomly placed in eight different teams that each developed an information system following the waterfall model and with various degrees of reuse. The differences in quality was assessed across four reuse categories; new, extensively modified, slightly modified, verbatim [44]. The results of the study showed that defect-density decreases linearly with the rate of reuse [45]. Melo et al. claim that this is strong evidence that reuse help improve quality across the covered reuse rate range, that is, from 0% to 64%. Furthermore, the results supported their assumption that reuse in object-oriented software development results in lower rework effort [45]. They claim that their results form the data analysis and conclusion can be generalized as follows.

Each additional 10 percentage points in reuse rate, within the reuse rate interval covered by our data set, decreases rework by nearly 8.5%.

- Melo et al. [45]

Their studies also conclude that increased reuse rate results in linearly increasing productivity and reducing effort [45]. They state as quoted below.

When there is no reuse, productivity should be expected to be around 14 SLOC per hour and each additional 10 percentage points in the reuse rate should increase productivity by 11 SLOC per hour.

- Melo et al. [45]

An Empirical Study of Software Reuse Benefits in a Large Telecom Product and An Empirical Study of Software Reuse vs. Defect-Density and Stability both document the same industrial case study of a large telecom company, Ericsson, performed by P. Mohagheghi. The studied components were developed in-house, made reusable through an extractive approach, and shared between two products in a product-family approach across two countries. However, An Empirical Study of Software Reuse Benefits in a Large Telecom Product supplements An Empirical Study of Software Reuse vs. Defect-Density and Stability with among other one hypothesis, lessons learned regarding metrics and data collection, a prestudy related to product line families and releases-based development and more focus upon software evolution.

Table 2.4 presents the hypotheses and the results of the study.

The study’s results conclude that reused components have lower defect-density than non-reused ones [40]. P. Mohagheghi et al. observed that the reused components had more defects of the highest severity before delivery than expected from the total
Table 2.4: Research hypotheses and results

<table>
<thead>
<tr>
<th>HypID</th>
<th>Hypothesis Text</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>Reused components have the same defect-density as non-reused ones.</td>
<td></td>
</tr>
<tr>
<td>HA1</td>
<td>Reused components have lower defect-density than non-reused ones.</td>
<td>Accepted</td>
</tr>
<tr>
<td>H2</td>
<td>There is no relation between number of defects and component size for all components.</td>
<td></td>
</tr>
<tr>
<td>H02-1</td>
<td>There is no relation between number of defects and component size for reused components.</td>
<td>Not rejected</td>
</tr>
<tr>
<td>H02-2</td>
<td>There is no relation between number of defects and component size for non-reused components.</td>
<td>Rejected</td>
</tr>
<tr>
<td>H3</td>
<td>There is no relation between defect-density and component size for all components.</td>
<td></td>
</tr>
<tr>
<td>H03-1</td>
<td>There is no relation between defect-density and component size for reused components.</td>
<td>Not rejected</td>
</tr>
<tr>
<td>H03-2</td>
<td>There is no relation between defect-density and component size for non-reused components.</td>
<td>Not rejected</td>
</tr>
<tr>
<td>H4</td>
<td>Reused and non-reused components are equally modified.</td>
<td></td>
</tr>
<tr>
<td>HA4</td>
<td>Reused components are modified more than non-reused ones.</td>
<td>Rejected</td>
</tr>
</tbody>
</table>

distribution, but fewer post-delivery defects[40]. This indicates that defects of reused components are given higher priority to fix[40].

There wasn’t observed any significant relation between either the number of defects and component size, or between defect-density and component size. Probably other factors, such as reusability, functionality and programming language can be related to components’ defect-density [40]. P. Mohagheghi et al. concluded that reused components are actually modified less than non-reused ones [40]. Based on these results, the study concluded that packaging shared functionality into reusable components reduces defect-proneness and improves stability of software.

Table 2.5: Research questions implied by P. Mohageghi in her study of Ericsson

<table>
<thead>
<tr>
<th>ID</th>
<th>Research question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>How to initiate a product family or reuse program?</td>
</tr>
<tr>
<td>Q2</td>
<td>Which processes or steps in development should be modified and how?</td>
</tr>
<tr>
<td>Q3</td>
<td>How to assess return-on-investment either in reduced time-to-market, reduced cost of development, or reduced risk?</td>
</tr>
<tr>
<td>Q4</td>
<td>What is the impact of release-based development on data collection?</td>
</tr>
<tr>
<td>Q5</td>
<td>Do results of assessments of one release have prediction value for future releases?</td>
</tr>
</tbody>
</table>

As mentioned above, An Empirical Study of Software Reuse Benefits in a Large Telecom Product is also a documentation of the study performed by P. Mohagheghi.
et al. at Ericsson. This article supplements the hypotheses formulated and tested as documented in *An Empirical Study of Software Reuse vs. Defect-Density and Stability* with a new hypothesis and 5 research questions. The research questions from this study are given in table 2.5 [31]. The definition of the hypothesis formulated in *An Empirical Study of Software Reuse Benefits in a Large Telecom Product* that was not documented in *An Empirical Study of Software Reuse vs. Defect-Density and Stability* is: Reused and non-reused components are equally affected by requirement modification. This hypothesis could not be rejected as the data that was needed to test it was too coarse-grained.

The list below presents the answers P. Mohagheghi et al. found to their research questions [31].

**Answer to Q1** P. Mohagheghi et al. found, like other studies, that management support, the definition of common goals and a common infrastructure, and experienced personnel were critical factors for the success of reuse [31]. Furthermore, the product family or reuse program was initiated by evolve the initial architecture to a layered one, and the company moved toward planning for reuse.

**Answer to Q2** The procedures of knowledge sharing had to be adapted to the infrastructure of the company. Knowledge sharing within a single location could be facilitated through informal communication and face-to-face meetings, while geographical distance requires more formal communication such as documents [31].

**Answer to Q3** The study showed that reused components had lower fault-density and code modification rate. Furthermore, reuse reduced risks and lead time of the second of the two products in the product line family. Also the risk and cost associated with moving the development geographically, was reduced because of reuse and standardisation [31].

**Answer to Q4** Inconsistencies in the data, lack of systematic data collection hindered this question to be answered [31].

**Answer to Q5** Less fault-density and code modification rate of reused components were verified over multiple releases. Total code modification rate and reuse rate were almost the same over releases and can thus serve as baselines to other projects or future releases.

Furthermore, P. Mohagheghi et al. document lessons learned during the study regarding metrics and data collection [31]. These are presented in table 2.6 from [31].

### 2.9 An empirical study of defect reports

Defect reports have been analyzed in several empirical studies evaluating quality of software products, software processes, amount of rework and maintainability of products. P. Mohaghegi and R. Conradi have compared different defect classification schemes and suggested a common terminology to design classifications schemes [46]. This section will present some results from this study. Defect data is one of the few direct measurements of software quality in addition to the actual maintenance and enhancement jobs.
Table 2.6: Lessons learned regarding metrics and data collection

<table>
<thead>
<tr>
<th>Problem domain</th>
<th>Problem description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metrics definition</td>
<td>The granularity of data was too coarse. Proper metrics, such as size of new and modified code between releases, and requirement changes between releases, for incremental development were missing.</td>
</tr>
<tr>
<td>Tools and routines</td>
<td>Fields in the Trouble Reports and Change Requests were missing. The tools did not provide facilities for search in SQL databases. There were difficulties with integrating data from several releases.</td>
</tr>
</tbody>
</table>

Several indirect metrics from defect reports can also be useful in quality assessment of software. There are no standards for defect-reporting systems, and often the defects are analyzed after the project is finished. The available data are not necessarily gathered with a specific purpose to assess the software quality. To measure the software quality in a post process way, the measurement goals have to be stated in a mixture from experience or research questions (top-down approach) and from a prestudy of historical data.

The article *Defect Reporting for maintenance and Quality Evaluation* presents three requirements for a good measurement system, identified by Chillarge [46].

**Orthogonality** to ensure that factors can be evaluated without confounding the effects on the response. This means that measured factors must be uniquely identified.

**Consistency** implying that the same scheme can be used in all stages of software development and verification.

**Uniform** across products.

If the value space for each attribute is small, classifications can be made easier and result in less error prone reports [46].

Generalizing empirical studies in this field is difficult, because of differences in organizations, between people and technical factors. Classification of attributes allows classification of studies, such as evaluating their purpose and evidence [46].

### 2.9.1 Work related to software evolution

We base our studies of software stability on the work and studies performed by M. Lehman’s and presented in section 2.6 generally, and Lehman’s Laws and guidelines for how to handle the Third and Eight Law specifically. Accordingly, we will base our measurement of software stability upon his suggestion, which is as the number of changes in each release (of a component). Furthermore, we will interpret our results in the light of the literature presented in section 2.6.

### 2.10 Research in Software Engineering

This chapter gives an overview of research approaches and strategies in software engineering, and is mainly based on [47]. In software engineering new methods and
processes is constantly growing and the field is rapidly in change. Empirical research is an important tool in evaluating new methods and processes. Our study will use an empirical research method, with focus ongoing an improved understanding and identify relationships between different variables.

There are two types of research paradigms that have different approaches to software engineering. Qualitative research is concerned with discovering causes noticed by the subjects in the study, and understand their view of the problem at hand. The subjects are the persons taking part in the empirical study in order to evaluate an object. Quantitative research is mainly concerned with quantifying a relationship or to compare two or more groups. The aim is to quantify a cause-effect relationship. An advantage is that quantitative data promotes comparisons and statistical analysis.

Qualitative and quantitative research can be combined and are complementary to each other. Quantitative studies is appropriate for testing effects of a treatment, and qualitative studies can identify beliefs and understandings to find out why the results from the quantitative investigation are as they are.

2.10.1 Empirical Strategies

There are three main types of strategies that may be carried out, depending of the purpose of the empirical investigation [48]:

Survey is an investigation method often performed in retrospect, when an object has been in use and should be evaluated. Data are mainly gathered through interviews or questionnaires, with a sample from the population to be studied. The results are then analyzed and generalized to the population of which the sample was taken.

Case Studies are used for monitoring projects, activities or assignments. Data is throughout the study collected for a specific purpose, and serve as material for statistical analysis. A case study is an observational study, and are appropriate when the investigator has little or no control. It is normally aimed at tracking a specific attribute or establishing relationships between attributes. Multivariate statistical analysis is often applied for this purpose.

Experiments are normally done in a laboratory environment, which provides a high level of control. The objective is to manipulate one or more variables and control all other variables at fixed levels. In the experiment, subjects are assigned to different treatments at random. "True experiments", with participants or objects chosen at random, are often difficult to perform in software engineering. Quasi-experiments, where participants were impossible to choose at random, can also give important results, but their threats to validity must be carefully investigated.

2.10.2 Measurement

To control an empirical study and to see the effects, we must be able to both measure the inputs in order to describe what causes the effect, and the output.

Measurement are defined in as the following.

"Measurement is a mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterize an attribute."
The word *metric* have to different meanings in association with software engineering. Software metric is used to denote the field of measurement in software engineering and metric is used to denote an entity which is measured. The mapping from an attribute to a measurement value can be made in many different ways, and each different mapping consist of locating the attribute on a scale. The most common scale types are the following [47]:

- **Nominal scale** is a classification of entities. The attribut is mapped into a name or symbol.
- **Ordinal scale** ranks the entities after an ordering criteria.
- **Interval scale** orders the values in the same way as the ordinal scale, but here the relative distance between two entities are also meaningful.
- **Ratio scale** can be used if there exists a meaningful zero value and the ratio between two measures are meaningful.

Some attributes are directly measurable, while others are derived from other measurements and are called indirect measures. An other way to classify measures is to consider the viewpoint of wich the measurement has been taken. In an *objective measure* there is noe judgement in the measurement value, and the value is only dependent on the object being measured. In a *subjective measure* the person making the measure is contributing with some kind of judgement, and the measure is both dependent on the object and the viewpoint. In software engineering, the objects interesting for measurement are divided into three classes, described in [50] as given below.

- The *process* describes activities needed to produce software
- The *products* are the artifacts, deliverables or documents resulting from a process activity.
- *Resources* are the objects needed for a process activity, such ass personell, hardware, software.

Within each of these classes we distinguish between internal attributes, that can be measured purely in terms of objects and external objects measured with respect to how the objects are related to other objects.

In empirical studies the scale types are important as the statistical analysis depends on them. The most powerful statistical analyses requires interval or ratio scales.

### 2.10.3 Variables

To draw conclusions from a dataset we must identify which relationships we want to investigate. *Independent variables*, are those variables that we can change and contoll and have some effect on the dependent variable. The *dependent variable* gives a measure of the effect of the treatment. Usually just one dependent variable is chosen, and it should therefore be direct output from the hypothesis. [51]
2.10.4 Data analysis

To be able to draw valid conclusions from the data gathered in an empirical study, the data have to be interpreted. First descriptive statistics may be used to describe and graphically present the dataset. The goal of descriptive statistics is to get a sense of how the data is distributed and can also be helpful in detecting false data points. Useful statistical dependency techniques are; measures of central tendency, measures of dispersion, measures of dependency and various graphical visualization techniques. Next statistical analysis could be used to get evidence for drawing conclusions. This procedure is described in the next section.

Statistical analysis

Hypotheses are the basis for statistical analysis in an empirical study. Hypothesis are stated formally, and analyses of data are used to reject the hypotheses if possible. In cases where the hypothesis could be rejected conclusions can be drawn based on the hypotheses testing under given risks. Two hypothesis have to be formulated. These are described below.

**A null hypothesis, H0:** This hypothesis states that there are no underlying trends or patterns in the objects of study, the only reasons for differences in our observations are coincidental. The analyzer want to reject this hypothesis with as high significance as possible.

**An alternative hypothesis, H1:** This is the hypothesis in favor of which the null hypothesis is rejected.

After data are collected, tests about the hypothesis can be carried out. In many cases it is possible to calculate the lowest possible significance, *p-value*, with which we could reject the null hypothesis. The different tests can be classified into parametric tests and non-parametric tests. Parametric tests are based on a model that presuppose a specific distribution, in most cases a normally distribution is assumed. The other requirement is that parameters must be measured on at least one interval scale. In difference from parametric tests, non-parametric-tests do not require the same type of assumptions. It is preferable to use parametric tests when the required conditions are fulfilled, because the power of this types of tests is generally higher. Non-parametric tests are more general, and can be used instead of parametric tests. The choice of test also depend on wether there is one or more samples of collected data.

The most familiar tests defined by Wohlin et al. are presented below.

**t-test:** This test is a parametric test, used to compare two sample means.

**Mann-Whitney:** This is a non-parametric alternative to the t-test. It is always possible to use this test instead of the t-test.

**Paired t-test:** The test are used when there are two samples from repeated measures, and and they are desirable to compare.

**Wilcoxon:** This is a non-parametric alternative to the paired t-test. It requires that it is possible to determine which of the measures in a pair that is the greatest and that the difference is possible to rank.

**ANOVA:** Analysis of variance can be used to analyze results where there are a number of factors and a number of samples.
Chi-2: This is a family of non-parametric tests that can be used when data are in the form of frequencies.

In the context of our study, ANOVA and Regression analysis is the most appropriate. We will therefore prioritize a further description of them.

ANOVA

ANalysis Of VAriance can be used to analyze results where there are a number of factors and a number of samples. In its simplest form, the test compares the variability caused by treatment and variability caused by random error. A typical design for using ANOVA, is one factor with more than two treatments. The test can then compare if the number of samples have the same mean value. The results from this test can be given in a table like 2.7 from [34].

<table>
<thead>
<tr>
<th>Source of Variation</th>
<th>Sum of squares</th>
<th>Degrees of freedom</th>
<th>Mean square</th>
<th>$F_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between Groups</td>
<td>$SS_{Treatment}$</td>
<td>a-1</td>
<td>$MS_{Treatment}$</td>
<td>$F_0 = \frac{MS_{Treatment}}{MS_{Error}}$</td>
</tr>
<tr>
<td>Within Groups</td>
<td>$SS_{Error}$</td>
<td>N-a</td>
<td>$MS_{Error}$</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>$SS_T$</td>
<td>N-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N is the total number of measurements and a is the number of samples. The criteria for rejecting the null hypothesis is $F_0 > F_{a,f_1,f_2}$. Where $F_{a,f_1,f_2}$ is the upper $\alpha$ percentage point of the F distribution with f1 and f2 degrees of freedom.

Regression analysis

Regression analysis could be used to see if there is a dependency between two variables. Regression means fitting the data to points in a curve. This analysis is useful when we expect the data to have a linear relationship. In this case we will show that a fitting line that minimizes the sum of the quadratic distances to each point makes a linear regression.

Pearson correlation coefficient gives an indication of a systematic relationship between the values in two data set. It measures the strength of a linear dependency between the two variables. The coefficient will have a value between -1 and +1, and the value 0 if there is no correlation. A must be 0.7 or higher to be considered as high.

R-square ($R^2$) is defined as the ratio of the sum of squares explained by a regression model. It show how much of the independent variable that is explained with the variation of the dependent variable. Adding more independent variables $R^2$ will cause the factor to increase. This also explain that a low value, will indicate that there are other factors that impact the dependent variable.

R-square ($R^2$) adjusted show the same as $R^2$, but is adjusted to the number of parameters in the model. A $R^2$ with value 0.8 means that 80% of the variation in the dependent variable is explained by the estimated regression line. The $R^2$ should have a value of 0.8-0.9 or higher to be considered as high.
The significance of the observed regression line is indicated by the *Probability value (p-value)*. We would prefer the p-value to be as small as possible to show that the coefficients are correct and that we have a good correlation. The p-value denote the probability that the linear relationship is just a coincidence.

Graphical visualization give a good overview of the dataplots. A *scatter plot* show how points (x,y) lies in a two dimensional coordinate system. By examining this we can see how outspread or concentrated the data points are and if there is a tendency of linear relation.

A *residual plot* show how the data point are distributed above or below the regression line. This could be a check that the plots are evenly distributed round the line.

### 2.10.5 Validity of results

A very important aspect of an empirical study is the validity of the results. Adequate validity is necessary for the results to be valid for the population of interest. The result should in the first place be valid for the population of interest, and in the next place the results might be generalized to a broader population. Wohlin defines four categories of validity threats [55]:

- **Conclusion validity** is concerned with the relationship with the treatment and outcome. We want to assure that there is a statistical relationship, with a given significance.

- **Internal validity** is about making sure that there if there is a causal relationship between the treatment and the outcome, this must be a causal relationship and not result of a factor not considered.

- **Construct validity** is concerned with the relation between theory and observation.

- **External validity** is concerned with generalization. If there is a causal relationship with the construct of the cause, and the effect, could the result be generalized outside the scope of the study?

When evaluating the validity, possible threats are identified. If there are any, they have to be addressed or accepted. Sometimes threats must be accepted, but then they must be handled when interpreting the result. Different studies claim different prioritization of validity types.
Chapter 3

State of the practice at Statoil

This chapter presents the Statoil context, gives a description of the SJEF architecture that contain building blocks we shall investigate in our study and data from the development project which we can use for analysis.

3.1 Statoil context

The environment of our study is the Statoil Java Enterprise Framework (SJEF) architecture at Statoil ASA. Statoil ASA is one of the largest operators on the Norwegian continental shelf, and has its interest in a number of oil and gas areas.

3.1.1 Statoil Java Enterprise Framework (SJEF)

Statoil is developing a new generation of information systems in order to support current and future central business areas. The systems should be easily adaptable and developed within a common architecture. The solution to this was SJEF, an architectural framework defined in 2004 [56].

Motivation for introducing SJEF in Statoil

The traditionally enterprise applications at Statoil were built with a tightly integrated user interface and with poor integration capability. The individual application had to some extent been built as the only application or system needed. Consequences of this, was reduced productivity and inconsistent information in the business process. The IT people saw the problem and initiated the task of integrating the standalone applications into consistent systems, resulting in better work processes.

SJEF is going to providing a set of mechanisms and tools that facilitates construction of cohesive and loosely coupled information systems that can be tailored to individual needs. The main business driver behind SJEF is to make a technical platform for development and integration of a new generation of Information and Communication Technology (ICT) solutions, that facilitates continuously improvement of the business processes. SJEF provides general building blocks and design principles valid for any business that need to build or integrate enterprise class systems. Other motivations for composing SJEF is to facilitate re-use of existing assets and keep the life-cycle cost more linear and predictable. [57].
3.1.2 Java Enterprise Framework (JEF)

JEF is a J2EE Technical Framework for Enterprise Applications. The SJEF architecture is implemented with this framework, and tailored specifically for the requirements in Statoil. Using an existing framework, with standards and common components, will provide benefits like decreasing development time of products, improve software quality and reducing cost and risks.

The JEF framework consists of several separate components, which can be applied separately or together when developing applications. Figure 3.1 from [58] and table 3.1 based on [59] gives an overview and description of the components in JEF.

![Figure 3.1: Overview of the Java Enterprise Framework](image)

3.1.3 The SJEF projects

Three projects at Statoil, Digital Cargo Files (DCF), Shipment and Allocation (S&A) and Contract and Verification (C&V), are initiated to develop applications with SJEF as the underlying architecture. The last two are put on hold until November 2005. The JEF components are developed by a team called the JEF Team consisting of 4 persons.

We hope that we can get data from the project that develops DCF, and use DCF as our example of a JEF-based system and analyze data from testing and error reporting in this project. The DCF application system is a document storage system and is meant to replace the current practice of cargo files, which are physical folders containing printouts of documents. Beside of being based on the SJEF architecture, DCF is integrated with other components. BizTalk, the currently integration
### Table 3.1: Components in JEF

<table>
<thead>
<tr>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEFClient</td>
<td>The JEFClient component is basically a large class library providing functionality. Other applications using this component, does not have to include lots of standard low level client coding. The main focus on the JEFClient is on widgets and their related data binding. This bring along easy connection of domain models and widgets. The widget will then display configured information from one or more domain objects.</td>
</tr>
<tr>
<td>JEFWorkbench</td>
<td>The JEFWorkbench component is an application providing features like authentication, authorization, navigation, preferences, plugins and more. This component enable login for users and present the activities authorized for each user. Together with the JEFSecurity component, JEFWorkbench provide role based, adaptable and secure access.</td>
</tr>
<tr>
<td>JEFUtil</td>
<td>This component provides different utilities such as service locators, jnlp launcher, logging (based on Apache Log4J), xml utils, Exception framework etc. jnlp launcher, logging(based on Apache Log4J), xml utils, Exception framework etc.</td>
</tr>
<tr>
<td>JEFSecurity</td>
<td>JEF Security provide both authentication and authorization services to code running on the client and on the server. Authentication for the client happens via the Lightweight Directory Access Protocol (LDAP) in Statoil. A protocol used to access a directory listing to enable lookup queries. Authorisation is provided on both the client and server. The information can be fetched from Role Based Access Control (RBAC) system, and is used to support security control on the client and the server.</td>
</tr>
<tr>
<td>JEFSession Management</td>
<td>JEFSessionManagement component provide communication between the client and the server. The component can support any protocol supported by Java, including HTTP/HTTPS. The server side services that will be available from the client can be configured.</td>
</tr>
<tr>
<td>JEFIntegration</td>
<td>Integration of components and information systems cause challenges like differences between applications in their; datastructures, programming language, communication protocols, operating platforms, and changes over time. Other challenges are related to unreliable or limited resources as slow or unstable networks and periods with system maintenance. A common solution to this challenges is integration by messaging. The JEFIntegration component provide communication between application components, information systems and other systems/applications in the overall architecture.</td>
</tr>
<tr>
<td>JEFDataAccess</td>
<td>The JEFDataAccess component provides data access functionality to the developers, so they can use a common pattern when creating Java Data Objects. This result in consistent data access functionality within applications and between projects.</td>
</tr>
</tbody>
</table>

infrastructure, will handle message passing to DCF. *Meridio*, a tool for managing documents, will support this also in DCF. [60].
3.2 Data from various projects

At today’s date, October 10th, we do not know for sure which data we will receive from the JEF Team at Statoil. This section presents

As mentioned in the last subsection, Statoil has planned three projects for developing applications that utilize the SJEF architecture. Today just one of the projects has registered data. The Digital Cargo Files (DCF) project started in February 2004, and the final release will be delivered during the autumn 2005. This is the reason why we at this point in time don’t know which data we will receive from the JEF Team. The following subsections describe the test stages and test planning in the DCF project.

3.2.1 Test stages

The DCF project has different test stages, which the outcome will give important information to give answers to some of our research questions. The DCF test stages are presented in [61], and are here supported with a description from [62].

- **Unit test**: Testing of units in the system.
- **Module test**: Testing individual modules of the system.
- **Integration test**: Testing the composition of modules.
- **System test**: Testing the whole system against the user documentation and requirements specification, after the integration testing is finished.
- **Usability test**: Testing the ability of users to find out and understand how to use the system.
- **Acceptance test**: Testing the system with support from the user organization of the system. The focus on this test is on usability and give the users opportunity to accept or reject the system.
- **Operational verification**: Test to determine if the delivered system meets the needs of its users and the expectations of its stakeholders after a period in use in its production environment.

3.2.2 Test planning

Plan for the system tests of JEF components are described in a test plan document from 12.04.2005, which this section will be based on. The overall testing objectives are to ensure that functional and non-functional requirements meets the initial Requirement Specification (RS), and the Change Requests (CR) and Incidents. The purpose of the tests are to verify that the new/updated JEF components perform as specified when integrated with other components. An Example application (XApp) will be set up with a graphical user interface with a main menu for all functionality. All developers will test their own environment.

The system/integration test stage will use the following test types. [63].

- Functional testing; fulfilling of requirements in RS and the initial CRs.
- Test of the user interface; comply to guidelines
- Non-functional testing;
- Performance testing; response time - detailed requirements to be provided in the test specification
- Reliability; force failures - failures should be handled to minimize the effect on users and data
- Repeatability; the same input should always provide the same response
- Integrity testing; comply to requirements of the initial RS and CRs.

Acceptance criteria

The pass of a specific test item is based on all the defined test cases, test scripts and other executed tests, the result and actions taken or that the test manager define the test as not applicable. This section is based on PLAN FOR SYSTEM TESTS, JEF components [63].

For a specific test in a test script to pass, the outcome of the test is compared with the expected result and approved by the test manager.

Criteria for a test script to pass, might be:

- All tests have been executed and no outstanding CRs, or
- All outstanding CRs being reviewed by test manager and service manager:
  - Maximum 0 errors with severity "Critical" in each JEF component
  - Maximum 0 errors with severity "High" in each JEF component
  - Maximum 1 errors with severity "Medium" in each JEF component
  - Maximum 2 errors with severity "Low" in each JEF component

A whole test stage is passed when all the planned tests are executed, the result evaluated and appropriate actions taken, or the test manager has defined the test as not applicable for the test stage.

The classification of errors is described in the test plan and will be summarized below:

Critical The defect causes system crash, loss or corruption of data. The defect stops the testing and the defect must be corrected or the change request must be implemented before the system can work properly. A defect with this grade could also mean that the system do not fulfill critical business functionality or will disrupt other systems.

High The defect means loss of a part of required functionality or quality. Alternatively the defect stops the current test or have impact on tests concerning interface units.

Medium The defect could mean loss of part of required functionality or quality, but there exist ways to work around the problems. The defect have only impact on this test, and in limited degree on the test progress. Correcting the defect will add value to the business like in time and/or costs.

Low The defect may only be cosmetic and has no impact of importance on the functionality and quality. The defect does not stop the test and does not block the test progress.

After the pass of a test stage, the test will be documented in a test report. This report should contain a summary of the test effort and recommendation of approval.
3.2.3 Test reports

We have inspected some Test reports from both the JEF system and the DCF system. Below we give a summary of them:

**FINAL TEST REPORT, JEF Forvaltning, Test stage: System test, 14.06.2005 release**

The JEF Forvaltning (Management in English) test is on the JEF-components in progress. An application, XApp, are implemented on top of JEF to test the functionality. The test shows that the XApp is the most defect-prone component, so the consequence of the result is to focus on setting up the test functionality in XApp for new functionality in JEF, at the same time as producing the new JEF functionality.

**FINAL TEST REPORT, JEF Maintenance, Test stage: System test, Release 01.09.2005**

This test is on the next release of JEF from the foregoing test, that has more added functionality. The shows that the XApp still is the most error intensive area.

**FINAL TEST REPORT, PROJECT Digital Cargo Files, Test stage: Integration/System Test, Rev1.4**

This document is the final test report for the System/Integration Test stage in Project Digital Cargo Files.

**FINAL TEST REPORT, PROJECT Digital Cargo Files ver 1.0, Operational Verification, Rev1.1**

The operational verification test has been an integrated part of the daily operation, and checks if the the needs of its users and the expectations of its stakeholders are met.

3.2.4 Trouble reporting

Everybody is responsible for submitting change requests within their area, but most of them are discovered during the testing. The reports can be handled at any time during the project, but must go through a full design/development/test cycle. An alternativ is to gather change requests and perform an extra design/development/test iteration. ClearQuest is used to manage all the changes.

3.2.5 Classification of changes

The JEF development project has defined to types of changes. Scope changes are enhancement request related to perfective maintenance, i.e. adding new functionality. Incidents are identified defects and errors, related to corrective maintenance, that leads to wrong performance of the system and need to be corrected. This definition is given in [64].

The sources from Statoil use different terms to describe the categories. Inspection of data in ClearQuest and examination with people at Statoil show that there are some inconsistencies with how the changes are divided into categories when they are reported. Reports on the pure JEF system are different from the reports in the DCF project. We are going to use data from the JEF system only, and this classification will be described here. Problems affecting the JEF system are either reported as incidents or as a changes. Incidents include this variations:

**Error** means the same as defect. The system does not follow the specification.

**Error in other system** means that the error will be corrected in relation to that system.
Duplicate is corrected as part of another error

Rejected means that the incident is not approved to be worked on

Postponed indicate that the incident is set on hold, needs more analyse or will be continued on later

Enhancement applies as a change. Enhancements could be moved to changes, but most of them are corrected as errors.

The data we have received did not include any classification attribute on the changes.

Workflow for scope changes

Figure 3.2 from [65], shows the workflow regarding registration and fixing of changes. When an enhancement is identified, the request must be registered in ClearQuest. The request will be handed over to an appropriate person, who will be responsible for the analysis. The analyse results in a specification including all effects like money, time and effect on other work. The specification is then approved to be performed and assigned to a project member. The project member checks out the file to work on, and stay in progress until the coding is finished. After the unit test is complete the code is deployed in the test environment and assigned to a tester. The testperson will perform a system and an acceptance test, and close the request if the test is passed.

Workflow for defects

Usually the test team identify and submit defects, but anybody discovering an defect can report it. Figure 3.3 from [65], shows the workflow for defect reporting, and will further be described. The report should be filled in with as much information as possible, including project name. The defect will then be analyzed, and acquired classification and prioritazion. The analysiz will involve seeking the root cause and area of the problem, and then be assigned to an appropriate developer. The
developer then check out a file and set the defect state to open. When the code and unit test is complete, the code is passed to the test environment and assigned to a tester. If the test fails, the problem will be re-assigned to the developer. If the test passes, the defect handling is complete and closed.

![Defect workflow](image)

Figure 3.3: Defect workflow

### 3.2.6 ClearCase

IBM Rational ClearCase is a software product that provides life cycle management and control of software development assets. ClearCase is used for configuration management in the projects related to SJEF. This software product contain libraries for storing project items. Project items are documents like architecture document, specification and design document, test plans, test specifications, test reports, system and user documentation. ClearCase will also hold source code and scripts. All final libraries and executables will also be hold in their home version control system. A general rule is that items should be checked into the system every day, to assure that no more than a days work will be lost in case of problems. The changes that are checked in must always be commentated.

### 3.2.7 ClearQuest

The SJEF project in Statoil use a powerful tool, called Rational ClearQuest, to help tracking information in their development process. Rational ClearQuest is a flexible defect and change tracking system that captures and manages all types of change requests throughout the development lifecycle, helping organizations quickly deliver higher quality software.

### Data in ClearQuest

Every scope change and incident are registered in ClearQuest by filling out information in attribute fields. We will use the word defect instead of incident in the rest
of the report, because this denote the same thing. For defects in JEFComponents the attributes for each change available to us are given in table 3.2

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>a unique identifier, for JEFcomponents on the form JEF-FXXXXXXXX (X is a number)</td>
</tr>
<tr>
<td>Headline</td>
<td>a free textual description of the problem</td>
</tr>
<tr>
<td>Priority</td>
<td>the level of which the problem is prioritized 0. Critical, 1. High, 2. Medium, 3. Low, 4. Not prioritised</td>
</tr>
<tr>
<td>State</td>
<td>indicates where in the workflow cycle this change is at a specific time. The state is updated every time it is transferred to a new state. The classification of values for this attribute is the same as in the workflow in figure 3.2 for changes and 3.3 for defects.</td>
</tr>
<tr>
<td>Classification</td>
<td>The various types of changes are described in 3.2.5. This attribute is not consistently registered, and there are a lot of blanks. Therefore there will be difficult to use it in an analyze.</td>
</tr>
<tr>
<td>Severity</td>
<td>indicates how critical the problem is. The scale of this attribute is described in section 3.2.2.</td>
</tr>
<tr>
<td>Estimate</td>
<td>number of hours estimated to fix the problem. This attribute has many blank fields, and would therefore be problematic to use in an analyze.</td>
</tr>
<tr>
<td>Remaining</td>
<td>number of hours remaining on the problem fixing. This attribute has many blank fields, and would therefore be problematic to use in an analyze.</td>
</tr>
<tr>
<td>Subsystem</td>
<td>in which subsystem the problem is located, i.e. for JEF, JEF components.</td>
</tr>
<tr>
<td>System</td>
<td>in which main system the problem is located, e.g. JEF, DCF etc.</td>
</tr>
<tr>
<td>history.action_name</td>
<td>indicates the states that the change are going through. The change is registered with a new line, every time it is transferred to a new state, and this attribute contains the new state. The classification of values for this attribute is the same as in the workflow in figure 3.2 for changes and 3.3 for defects.</td>
</tr>
<tr>
<td>history.action_timestamp</td>
<td>the date and time when the change is submitted or transferred to a new state. The format of this attribute is dd.mm.yyyy hh:mm</td>
</tr>
</tbody>
</table>
Chapter 4

Research agenda

This chapter presents the research agenda of our study. It starts with a description of our interests and the research questions we would like to focus on. Then the metrics that we need and our the hypotheses is given. This is followed by the a description of the research method to be used to test the hypotheses.

4.1 Introduction

As described in the chapter 2, software reuse is used to reduce time-to-market and achieve better software quality, and software evolution is a phenomenon that can be a threat to the quality of the software product. There have been a number of studies of software quality, and several researchers have put their focus upon attributes such as fault-density. There have also been studies on the impact software evolution and release-based development have on software quality. However, few studies have looked at software quality in the context of reuse, and even fewer have simultaneously considered software evolution.

4.1.1 Overall research goal

Our overall research goal is to analyze the defect and change history of the JEF-components developed by the JEF Team at Statoil in order to assess the software quality and evolution of reusable components from the JEF Team at Statoil ASA. Our quality focus will be defect-density and stability (as change-density).

4.1.2 Terminology

This subsection describes how we define the expressions used in the following subsections.

KLOC means Kilo Lines Of Code and is the metric used to describe a module’s size.

Defect will be used in stead of faults, errors or failures, and we will not distinguish between active or passive faults or human or machine origin of these. This is consistent with who [10] uses this term.

Defect-density will we define as the number of faults or defects in the software module divided by the size of a same module, which is the definition used by [10].
DR Defect report, an entry in a defect reporting scheme.

Defect severity defines how difficult it is to fix the defect.

Change is concerned with modification of code. In accordance to 40.

Change-density is the number of changes of the software component divided by the size of this component.

Change history includes both functional changes and perfective, adaptive, preventive and corrective changes.

Change rate is defined as the change-density divided by the total number of days between the number of days between the first and last change.

CR Change report, an entry in a defect reporting scheme.

JEF-components refer to the components developed by the JEF Team at Statoil ASA on the basis of the Java Enterprise Framework, for reuse in among other the DCF and S&A application systems.

4.2 Formulation of research questions

The formulation of our research questions was a quite time-consuming process. We started the search for suitable research questions September 18th and landed on the final set November 22nd 2005.

Our basis was the research goal stated in section 4.1.1 and on the master thesis of Ole Morten Killi and Henrik Schwarz 38, the technical report of Parastoo Mohagheghi 40 and the unpublished paper 68. Furthermore, we consulted the SEVO project 24 to discuss and validate our alternative research questions to assess their relevance to the research community and related work.

In September 2005 it was still not clarified what data we would receive from Statoil. Before the data could be retrieved from Statoil, we had to define our research questions and the associated metrics. Furthermore, the data set that we would receive from Statoil would be restricted by Statoil’s openness and any potential confidentiality associated with their data and reports on defects and changes, as well as the availability of such reports and whether these reports have records for the data we need.

We hoped that we would receive data from Statoil that we could use to compare the quality of the reusable components with a baseline. The list below shows how we characterized and classified the possible comparisons.

JEF vs. the top-level applications This kind of comparison is not quite legitimate because objects of comparison is not of the same nature. The two systems are built in different ways with different purposes and assess different properties. Therefore, we question if they are comparable.

JEF vs. non-reusable middleware This is a legitimate comparison, but if Statoil haven’t recorded and stored the data needed from earlier middleware components, the data will be lost and not retrievable. This will make it impossible to carry out the comparison.

JEF-based systems vs. earlier systems This comparison compares systems composed by JEF-based application system with a system composed by non-reusable
components only. This comparison holds the same properties as the comparison of JEF and a non-reusable middleware. In addition to the results given by that comparison, this one will reveal the consequences of using reusable middleware on the top-level application component and the system as a whole, and not just the middleware. This because data on both the top-level application component and the total system is considered.

A serie of releases of a JEF-based system Such a study can reveal interesting details on the software evolution of the system. E.g. it can reveal whether defect-prone modules continue to be defect-prone in later releases. The comparison is legitimate because one release of the system is compared to another release of the same system. Therefore, the units of comparison is of the same kind.

4.2.1 Initial set of research questions

We decided to use an top-down and bottom-up procedure in order to identify the set of research questions that would both reflect our interests and be answered by the data received for Statoil. First, we used the GQM procedure on the basis of our goal formulated in section 4.1.1 to come up with interesting research questions and the needed metrics we needed to answer these. This resulted in the initial set of research questions, metrics and environment characterization. The set of research questions were tested for relevance in cooperation with the SEVO project before we handed them and their metrics over to A.A. Gupta, who was the intermediary consultant between us and Statoil.

Table 4.1 shows the initial set of research questions sorted according to quality focus. The research questions' identifiers are formulated "FRQID<X><y>". The prefix F stands for first. The prefix is needed to distinguish this set of research questions from the two following ones. <X><y> is a number succeeded by a letter. The number is the unique identification of the subset of research questions, and the letter identifies the research question uniquely within the subset it belongs to.

<table>
<thead>
<tr>
<th>Quality focus</th>
<th>ID</th>
<th>Research question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defects</td>
<td>FRQ1a</td>
<td>Do reusable components that have high defect-density before/after delivery in one release remain defect-prone in later releases?</td>
</tr>
<tr>
<td></td>
<td>FRQ1b</td>
<td>Do reusable components that have high defect-density before delivery, have high defect-density after delivery?</td>
</tr>
<tr>
<td></td>
<td>FRQ2a</td>
<td>What is the defect-density and defect severity in JEF-based systems during testing?</td>
</tr>
<tr>
<td></td>
<td>FRQ2b</td>
<td>What is the defect-density and defect severity in JEF-based systems in field use?</td>
</tr>
<tr>
<td></td>
<td>FRQ2c</td>
<td>What is the defect-density and defect severity in systems not based on JEF-components in testing?</td>
</tr>
<tr>
<td></td>
<td>FRQ2d</td>
<td>What is the defect-density and defect severity in systems not based on JEF-components in field use?</td>
</tr>
<tr>
<td>Stability</td>
<td>FRQ3</td>
<td>What is the change rate in JEF-based systems as opposed to systems not based on JEF-components?</td>
</tr>
</tbody>
</table>

Table 4.2 presents the metrics we would need to find the answers to these research questions.
Table 4.2: Metrics

<table>
<thead>
<tr>
<th>ID</th>
<th>Related RQ</th>
<th>Metric description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>FRQ1 (both), FRQ2(all)</td>
<td>Number of defects</td>
</tr>
<tr>
<td>M2</td>
<td>FRQ1 (both), FRQ2(all), FRQ3</td>
<td>Module size</td>
</tr>
<tr>
<td>M3</td>
<td>FRQ1 (both), FRQ2(all)</td>
<td>Defect-density (calculated from M1 and M2)</td>
</tr>
<tr>
<td>M4</td>
<td>FRQ2 (all)</td>
<td>Defect severity</td>
</tr>
<tr>
<td>M5</td>
<td>FRQ3</td>
<td>Number of changes</td>
</tr>
<tr>
<td>M6</td>
<td>FRQ3</td>
<td>Date of change</td>
</tr>
<tr>
<td>M7</td>
<td>FRQ3</td>
<td>Change rate (calculated from M5, M6 and M2)</td>
</tr>
</tbody>
</table>

Motivation for the research questions

This section explains the motivations of the different research questions in Table 4.1. In addition to the motivation given below, comes the general motivation of establishing a baseline that be used by Statoil to identify trends in their software process, and identify and exploit the feedback mechanisms and controls that apply to Statoil’s software process. Furthermore, such a baseline can be used by future studies for comparison with trends in other projects or companies. This baseline, and additional studies of other companies, can hopefully be used to identify today’s trends regarding the impact of reuse on software quality.

**FRQ1** As described in the prestudy, defect-density is used as an indicator of software quality in among others P. Mohagheghi et al. [40]. They state that in order to verify that defect-density is a good quality indicator, we must assess whether defect-prone components stay defect-prone after release, and in several releases, and build a prediction model [40]. An answer to FRQ1a that has a satisfactory confident level will make it possible to build such a prediction model valid for software developed under the same conditions of those of the DCF and S&A projects. Furthermore, research questions like FRQ1 is sought after by among others P. Mohagheghi and R. Conradi [31], and its result can be useful in association with Bennett and Rajlich’s version staged model [28].

Furthermore, [12] states that high fault-density before delivery may be a good indicator of extensive testing rather than poor quality. [40] follows this statement up by saying that fault-density cannot be used as a de-facto measure of quality, but remaining faults after testing will impact reliability. Thus, she proceeds, it is equally important to assess the effectiveness of the testing phases, and build prediction models [40]. This is our basis for formulating FRQ1b. By assessing defect-density of the reusable components before and after delivery, we can get measure the effectiveness of the test phases of the DCF and S&A projects. If this result has a satisfactory confidence level, it can be used as a prediction model for defect-density in reusable components developed by the DCF and S&A projects, after delivery.

If our results are supported by studies in other projects and companies, it might be possible to generalize our prediction models to projects external to the DCF and S&A projects and Statoil.
CHAPTER 4. RESEARCH AGENDA

FRQ2 This research question studies defect-density and defect severity across two dimensions; reusable versus non-reusable components, and during testing versus in field use. Assessment of these attributes during testing and in field use for both reusable and non-reusable components will give us the following research opportunities.

a To compare defect-density and defect severity in reusable and non-reusable components during testing and in field use. As discussed below, it is interesting to see if we get similar results as [40].

b Possibility to identify a pattern regarding these attributes for reusable and non-reusable components. If the results have a satisfactory confidence level, we can build prediction models for defect-density and defect severity for both reusable and non-reusable components developed by the two projects. Such prediction models can be very helpful to the projects, and for other projects or even companies if their external validity can be sufficiently verified.

We want to investigate the defect-density and defect severity of reused and non-reused components during testing and in field use to see if we get results that are comparable to those found by P. Mohagheghi et al. in their study of Ericsson Grimstad. As stated in [40], the external validity of the results of this study are threatened by the fact that the entire data set is taken from one company. Furthermore, the data set consists of a non-random sample of defect reports, and all components of a single release of one product [40]. As discussed in section 5 our results will be vulnerable to the same threats. If we get similar results as P. Mohagheghi et al., the possibility that the results given in their report have external validity is increased.

FRQ3 RQ3 is defined in order to investigate the stability of reusable and non-reusable components generally, by addressing the change rate in JEF-based application systems and systems not based on reusable components specifically. Whether our results can be generalized, is discussed in section 5. P. Mohagheghi et al.’s studies of Ericsson Grimstad concluded that the reused components was less modified than the non-reused ones [40]. Accordingly, they are less exposed to uncontrolled software evolution that may have negative ripple-effects on the software quality [6].

It is of our interest to do a similar study regarding the stability of reusable and non-reusable components in the DCF and S&A project at Statoil ASA. As discussed above, and further discussed in section 5, the external validity of both our study and the study performed by Mohagheghi is threatened. Similarities in the results of the two studies may indicate that they can be further generalized.

4.2.2 Research questions anno October 26th 2005

After representatives from the SEVO project had visited Statoil and had meetings with Statoil’s development department, it came apparent that we would not receive data that could be used to issue measurements on systems other than those based on JEF-components. That is, we would not receive data on the DCF and S&A application system components. The reason was that defects and changes in the older systems that was not based on the Java Enterprise Framework technology, had not been recorded in the same way as those in JEF-based application systems. Therefore, we had to exclude all research questions regarding systems that weren’t based on JEF-components. This left us with the comparison of JEF-based application systems across a series of releases as the only legitimate and possible comparison.
Table 4.3 shows the resulting set of research questions. These research questions’ identificator is prefixed by the letter S. This letter stands for second and it signalizes that this is the second set out of three sets of research questions. The rest of the identificator is as described for the first set of research questions.

<table>
<thead>
<tr>
<th>Quality focus</th>
<th>ID</th>
<th>Research question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defects</td>
<td>SRQ1a</td>
<td>Do reusable components that have high defect-density in the first release remain defect-prone in later releases?</td>
</tr>
<tr>
<td></td>
<td>SRQ1b</td>
<td>Do reusable components that have high defect-density before delivery, have high defect-density after delivery?</td>
</tr>
<tr>
<td></td>
<td>SRQ2a</td>
<td>What is the defect-density and defect severity in JEF-based systems during testing?</td>
</tr>
<tr>
<td></td>
<td>SRQ2b</td>
<td>What is the defect-density and defect severity in JEF-based systems in field use?</td>
</tr>
<tr>
<td>Stability</td>
<td>SRQ3</td>
<td>What is the change rate in JEF-based systems?</td>
</tr>
</tbody>
</table>

### 4.2.3 Final set of research questions

After further literature study and studies of the data we received from Statoil and the research questions we landed on October 26th, we reformulated our set of research questions one last time.

The final set of research questions is presented in table 4.4. The research questions in this set don’t have any prefix to their identificator. We want distinguish these research questions clearly from the above and make them easier to refer to and remember in the proceeding text.

<table>
<thead>
<tr>
<th>Quality focus</th>
<th>ID</th>
<th>Research question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defects</td>
<td>RQ1</td>
<td>How does the defect-density in reusable components evolve over several releases?</td>
</tr>
<tr>
<td>Stability</td>
<td>RQ2</td>
<td>How does the change-density in reusable components evolve over several releases?</td>
</tr>
<tr>
<td>Stability</td>
<td>RQ3</td>
<td>Are the reusable components with high change-density also the most defect-prone components?</td>
</tr>
</tbody>
</table>

**Motivation for change of research questions**

This set of research questions is smaller, but it still covers the most important aspects of the sets presented earlier. These research questions are formulated in a more general way, which better addresses and answers our overall research goal. Furthermore, the results of these research questions can hopefully be used as a prediction model in development of the new releases of the studied JEF components and other products developed in the same context.
RQ1: How does defect-density in reusable components evolve over several releases?

This research question is related to SRQ1, SRQ2 and SRQ3 in the set of research questions shown in table 4.3. It can be classified as a common and generalized version of the three. The result of this question will reveal the trend that SRQ1 and SRQ2 would indicate, generally, and that SRQ3 would state for JEF-based application systems, specifically. We did not have enough time to obtain the needed data, and interpret and analyse these as needed in order to answer SRQ1, SRQ2 and SRQ3. This is why we redefined these three questions into RQ1.

Chapter 2 presented studies that have focused on defect-density. Mohagheghi et al., Melo et al. and Graves et al. are among the few that have done this in the context of reuse [40] [31] [44] [20]. Their results conclude that software reuse decrease defect-density [40] [31] [44], and that change history is the best model for prediciting future faults [20]. Furthermore, P. Mohagheghi and R. Conradi imply questions regarding wether results of assessments of one release can be used for prediction of future releases, how we can compare the quality of releases or components in different releases, and how much components are modified between releases [31].

RQ2: How does the change-density evolve over several releases?

This research question is a pursuance of SRQ3 in table 4.3. The studies performed by P. Mohagheghi et al. at Ericsson examined hypotheses regarding the code modification rate, or degree of modification, in reused and non-reused components [31] [40]. Their results indicates that reused components are more stable than non-reused ones. This indicates that the probability of critical software evolution is lower for reuseable components. However, the study doesn’t focus upon the software evolution the same way as we wish to do, as its results are of a static manner.

Graves et al. studies change history of modules in order to build a prediction model for future faults [20]. The study concluded that the model best fitted to prediction is not size or complexity metrics, but the modules’ change history, including number of changes, length of changes, and time elapsed since the change. Both this study and Lehman’s article *Rules and Tools for Software Evolution Planning and Management* supports our definition of change-density and our study of this attribute in order to produce good predictions on the evolution of reusable components [20] [6]. Also P. Mohagheghi and R. Conradi, who are participants in The SEVO Project at NTNU [24], express an interest for an empirical study of maintenance across releases [31].

Lehman suggest that the number of changes counted per release is a good basis for the establishment of baselines in association with the identification of feedback mechanisms and controls in the feedback system that the software process is a part of [6]. Accordingly, our results can be used in the identification and following-exploitation of these mechanisms and controls that affect the JEF Team and its software process.

Furthermore, the results can hopefully be used to verify of those of P. Mohagheghi et al. at Ericsson and described in [40] and [31].
RQ3: Are the reusable components with high change-density also the most defect-prone components?

SRQ3 in table 4.3 was dropped in advantage of RQ3 because we want to explore a potential relationship between change-density and defect-density. The result of this research question will depend heavily on those of RQ1 and RQ2. We haven’t identified studies that study this relation specifically. Under the same conditions as the ones stated for RQ1 and RQ2, the result of this research question can be very useful to the JEF Team in the prediction and planning of future work on future releases of the studied components or new products.

4.3 Research Strategy

This section will briefly describe our research process and research methods used. There are three different types of empirical investigation methods as explained in section 2.10.1. Our empirical study can roughly be seen as a case study. We have carried out an observational study of the development of JEF components at Statoil. A case study should principally take place on an ongoing project or activity. We have collected data from the defect reporting, but data were handed to us, after the process had taken place. To be a genuine case study, we should have monitored the defect reporting.

Our process started with studying literature within the subject of software reuse, component based development and software evolution. Later we started to gather information about the research context, the SJEF architecture and the trouble reporting with JEF components. This gave us an overview of theories and findings from earlier studies and made us see opportunities of a possible research agenda for our study.

After getting more knowledge about the domain and the context, we started to consider different research questions. The method for making decision about research questions have been in both a top-down and a bottom-up approach. The top-down approach involved finding interesting goals and studies in the literature, and made us narrow our focus. The bottom-up approach was done by investigating available data and discover research opportunities. With this to approaches parallel in progress we found relevant questions for the subject which we could answer with the available dataset.

We have simplified the research questions into hypothesis. The hypothesis is a statement about relationship between different attributes in the research context. We identified independent and dependent variables from the attributes in the dataset which we could test relationships and make statements about the hypothesis. Chapter 5 describe our analysis and testing of the dataset.
Chapter 5

Results

In this chapter we will seek to find answers to the research questions presented in section 4.2.3. We will state hypothesis for this questions and use statistical tests to see if we can reject the hypothesis and draw a conclusion. We will discuss the causes of our results and their threats to validity.

5.1 Raw data

Our dataset consists of DefectReports and ChangeReports tracked in the software-program ClearQuest. Extracted data from ClearQuest were given to us in Microsoft Excel format. The time for extraction of data was 25.11.2005. We got two .xls documents at hand, one with reported incidents for JEF components and one with reported changes for JEF components. The registering of incidents span over the period 09.11.2004 to 14.11.2005 and the changes from 08.11.2004 to 14.11.2005. The JEF components have in this period gone threw 3 releases, and 3 test periods. The JEF releases, time of release and the timespan of work is summarized here:

- Release minor; 14.06.2005; 157 workdays, hollidays not concidered
- Release 3.0; 09.09.2005; 63 workdays; hollidays not concidered
- Release 3.1; 18.11.2005, 46 workdays; hollidays not concidered

We used Microsoft Excel as a tool to count changes and incidents for each JEF component. We divided the dataset after periods. Reports from before and including 14.06.2005 serve as material for Release minor, data from c serve as material for Release 3.0 and reports from 10.09 until 14.11.2005 goes under Release 3.1.

For both RQ1 and RQ2 we need the size of the components. #LOC(Lines Of Code) from the components in each release are shown in table 5.1.

5.2 Data analysis

In this section we will try to answer our reasearch qustions with basis in the material presented in the last section 5.1 We will set up hypothesis, use statistical tests , discuss the results and draw conclusions.
Table 5.1: Size of JEF components, in #LOC

<table>
<thead>
<tr>
<th>Component</th>
<th>Release minor</th>
<th>Release 3.0</th>
<th>Release 3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEFclient</td>
<td>7871</td>
<td>8400</td>
<td>8885</td>
</tr>
<tr>
<td>JEFdataaccess</td>
<td>181</td>
<td>181</td>
<td>268</td>
</tr>
<tr>
<td>JEFintegration</td>
<td>958</td>
<td>958</td>
<td>958</td>
</tr>
<tr>
<td>JEFsecurity</td>
<td>1588</td>
<td>1593</td>
<td>2374</td>
</tr>
<tr>
<td>JEFutil</td>
<td>1312</td>
<td>1359</td>
<td>1647</td>
</tr>
<tr>
<td>JEFworkbench</td>
<td>4187</td>
<td>4515</td>
<td>4748</td>
</tr>
</tbody>
</table>

5.2.1 Defect-density

We would like to investigate the defect-density in reusable components, and verify how the measure evolve threw different releases of components. We will seek to answer the research question:

**RQ1:** How does the defect-density in reusable components evolve over several releases? The answering method is to formulate hypothesis and test whether this could be rejected.

Dataset

We sorted the defect reports after which JEFcomponent they was related to. The reports contain one element for each state a defect have gone threw. We decided to count the submit element of the defects only. Table 5.2 show the defect-density, calculated as #DR/KLOC.

Table 5.2: Defect-density in JEF components, #DR/KLOC

<table>
<thead>
<tr>
<th>Component</th>
<th>Release minor</th>
<th>Release 3.0</th>
<th>Release 3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEFclient</td>
<td>17.1516</td>
<td>1.5476</td>
<td>0.1190</td>
</tr>
<tr>
<td>JEFdataaccess</td>
<td>11.0497</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>JEFintegration</td>
<td>3.1315</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>JEFsecurity</td>
<td>5.6675</td>
<td>0.6277</td>
<td>0.0000</td>
</tr>
<tr>
<td>JEFutil</td>
<td>1.5244</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>JEFworkbench</td>
<td>3.8214</td>
<td>0.8859</td>
<td>0.2106</td>
</tr>
</tbody>
</table>

Hypothesis

H10: The defect-density in JEFcomponents do not change over several releases.

H1A: There is a difference in defect-density for JEFcomponents in different releases.

Statistical method

The independent variable is release(timeperiod), and the dependent variable is defect-density. We have three treatments of the independent variable(release), and therefore we decided to use an ANOVA test. This test is parametric, but is robust enough to use without proving a normal distribution. We calculated the average defect-density for each release, see table 5.3. With the ANOVA test, we would like to see if there was a significant different in the mean-values of the releases.
Results

Table 5.3 summarizes the results from the ANOVA test.

Table 5.3: Average defect-density in each release

<table>
<thead>
<tr>
<th>Groups</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release minor</td>
<td>7.057677527</td>
</tr>
<tr>
<td>Release 3.0</td>
<td>0.510216868</td>
</tr>
<tr>
<td>Release 3.1</td>
<td>0.054943769</td>
</tr>
</tbody>
</table>

Table 5.4: ANOVA table for testing H1

<table>
<thead>
<tr>
<th>Source of Variation</th>
<th>Sum of squares</th>
<th>Degrees of freedom</th>
<th>Mean square</th>
<th>$F_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between Groups</td>
<td>184,229,589,5</td>
<td>2</td>
<td>92,114,794,75</td>
<td>7.748,81,4662</td>
</tr>
<tr>
<td>Within Groups</td>
<td>178,313,972,1</td>
<td>15</td>
<td>11,887,598,14</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>362,543,561,6</td>
<td>17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3 show that the average defect-density decrease with time. The ANOVA test we performed gave a $F_0$ value of 7.748,81,4662. Microsoft Excel computed the $F_0$, 0.005, 2, 15 = 3.682,320,344 with a P-value = 0.004,883,024. Since $F_0 > F_{0.005,2,15}$, it is possible to reject the null hypothesis at the level of the p-value. This give a 0.49% probability that the results are just coincidental.

Discussion

The investigation and analysis of the data agree. There is therefore not much to discuss related to the dataset and the analysis.

Conclusion

We reject the null hypothesis, and conclude that there is a difference in defect-density for JEF components in different releases.

5.2.2 Stability

Research has demonstrated that the stability of reusable components is good and do improve with time. As explained in chapter 4, we have chosen to use change-density as an indication of the stability, and see how this measure evolve over different releases. What we would like to answer is:

RQ2: How does the change-density in reusable components evolve over several releases?

Dataset

The change reports was sorted by JEF component and then counted per release. Table 5.5 show the change-density, calculated as #CR/KLOC.
Table 5.5: Changes per component size (KLOC) in JEF components

<table>
<thead>
<tr>
<th>Component</th>
<th>Release minor</th>
<th>Release 3.0</th>
<th>Release 3.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEFclient</td>
<td>13,4672</td>
<td>0,8333</td>
<td>0,2251</td>
</tr>
<tr>
<td>JEFdataaccess</td>
<td>0,0000</td>
<td>0,0000</td>
<td>11,1940</td>
</tr>
<tr>
<td>JEFintegration</td>
<td>3,1315</td>
<td>1,0438</td>
<td>0,0000</td>
</tr>
<tr>
<td>JEFsecurity</td>
<td>9,4458</td>
<td>1,8832</td>
<td>0,6072</td>
</tr>
<tr>
<td>JEFutil</td>
<td>4,5732</td>
<td>0,7358</td>
<td>0,0000</td>
</tr>
<tr>
<td>JEFworkbench</td>
<td>8,3592</td>
<td>1,1074</td>
<td>0,0000</td>
</tr>
</tbody>
</table>

Hypothesis

H20: The change-density in JEF components do not change over several releases.
H2A: There is a difference in change-density for JEF components in different releases.

Statistical method

The independent variable is release, and the dependent variable is change-density. As with RQ1 we have three treatments of the independent variable (release), and did also here decide to use an ANOVA test, to see if there was a significant difference in the mean-values of the releases.

Results

Table 5.6: Average change-density data

<table>
<thead>
<tr>
<th>Groups</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release minor</td>
<td>6,496150593</td>
</tr>
<tr>
<td>Release 3.0</td>
<td>0,933944788</td>
</tr>
<tr>
<td>Release 3.1</td>
<td>2,004382145</td>
</tr>
</tbody>
</table>

Table 5.7: ANOVA table for testing H2

<table>
<thead>
<tr>
<th>Source of Variation</th>
<th>Sum squares</th>
<th>Degrees of freedom</th>
<th>Mean square</th>
<th>$F_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between Groups</td>
<td>104,5199067</td>
<td>2</td>
<td>52,25995335</td>
<td>3,539521151</td>
</tr>
<tr>
<td>Within Groups</td>
<td>221,4704382</td>
<td>15</td>
<td>14,76469588</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>325,9903449</td>
<td>17</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6 show the average change-density over time. Release 3.0 have lower change-density than in Release 3.1, indicating that the change-density might not decrease with time. Results from the ANOVA test performed are given in table 5.7. The ANOVA test we performed gave a $F_0$ value of 3,539521151. Microsoft Excel computed the $F_{0.055,2,15} = 3,682320344$ with a P-value = 0.055059556. $F_0 < F_{0.055,2,15}$, and therefore it is not possible to reject the null hypothesis at a 5% significance level.
CHAPTER 5. RESULTS

20th December 2005

Discussion

The result indicate that the number of changes in the components do not change over the studied releases. If we inspect the data in table 5.5 we can see that for all components the change-density is higher in the descendant release, except for JDataaccess. The value JDataaccess have in Release 3.1 differ considerably compared to the other results. This might have an specific explanation. We will discuss this further in section 5.3.

Conclusion

We can not reject the null hypothesis.

5.2.3 Stability vs. Defect-density

We thought it would be interesting to see if there was a relationship between the stability and defect-density of components. More precisely if the stable components introduced less defects, by coupling the two datasets from the two tests performed. This could be an indication that changes in a component would lead to defects. To see if the data set that we have conducted support this, we seek to answer the question:

**RQ3:** Are the reusable components with high change-density also the most defect-prone components?

Dataset

To study the relationship between change-density and defect-density we computed the mean-values for each component over the entire period for the data registration.

<table>
<thead>
<tr>
<th>Component</th>
<th>Change-density</th>
<th>Defect-density</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEFclient</td>
<td>4.8419</td>
<td>6.2727</td>
</tr>
<tr>
<td>JEFdataaccess</td>
<td>3.7313</td>
<td>3.6832</td>
</tr>
<tr>
<td>JEFintegration</td>
<td>1.3918</td>
<td>1.0438</td>
</tr>
<tr>
<td>JEFsecurity</td>
<td>3.9787</td>
<td>2.0984</td>
</tr>
<tr>
<td>JEFutil</td>
<td>1.7697</td>
<td>0.5081</td>
</tr>
<tr>
<td>JEFworkbench</td>
<td>3.1555</td>
<td>1.6393</td>
</tr>
</tbody>
</table>

Hypothesis

H30: There is no relation between change-density and defect-density for the components.
H3A: There is a relation between change-density and defect-density for the components.
Statistical method

First we calculate the Pearson correlation coefficient to see how the possibilities for a linear dependency exists. Then we study the relation graphically, and then perform a regression analysis. A scatter plot with change-density on the x-axis and defect-density on the y-axis is shown in figure 5.1. A point in the diagram is a visualization of where a component lie in this coordinate system.

We also performed an regression analysis to determine if the defect-density is a function of change-density. Excel have a number of regression tools. $R^2$ and adjusted $R^2$ can be used to show how much of the x-values that is explained with the y-values. Figure 5.2 visualises the residual plot, and table 5.10 summarizes the result of the regression analysis.

Results

Table 5.9 show the result from computing the Pearson correlation coefficient. The value 0.849760606 in the left bottom corner is the coefficient. The correlation coefficient is high, indicating that there are good chances for the variables to have a linear dependency. As stated in section 2.10.4, the value is over 0.7 and can be considered as high. The scatter plot visualizes the relationship between the two variables. We can from this imagine that a line probably could fit between the points.

<table>
<thead>
<tr>
<th>defect-density</th>
<th>change-density</th>
</tr>
</thead>
<tbody>
<tr>
<td>defect-density</td>
<td>1.0000000000</td>
</tr>
<tr>
<td>change-density</td>
<td>0.849760606</td>
</tr>
</tbody>
</table>

Figure 5.1: Relation between change-density and defect-density
The residual plot in 5.2 show how points are distributed on both sides of the regression line. The point are evenly distributed above and below the line, with three points on each side, and so the regression analysis is valuable.

![Residual plot]

Figure 5.2: Residual plot between change-density and defect-density

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjusted R Square</td>
<td>0.65261636</td>
</tr>
<tr>
<td>P-value</td>
<td>0.032162221</td>
</tr>
</tbody>
</table>

Table 5.10: Summary of the regression results

The adjusted $R^2$ has the value 0.65261636, which means that 65.3% of the defect-density values could be accounted for by the regression line. A possible explanation of $R^2$ is that there are other independent variables that have impact on the distribution of change-density. Adding more independent variables would cause a grow of $R^2$. The P-value show the significance of the observed regression line. The P-value for the regression line is 0.032162221. This means that the probability of a random correlation is low. We could claim that a linear dependency between change-density and defect-density exist with a probability of, $(1-0.032)=0.967837779 = 96.78\%$.

Discussion

The different statistical methods more or less contradict the hypothesis in several ways. *Pearson correlation coefficient* = 0.849760606, give a strong implication of a linear dependency between the two variables. The same does the p-value = 0.032162221, of the regression analysis. The case that the points in the regression
plot are evenly distributed, verify that the analysis is valuable. A relatively low value on \( R^2 = 0.65261636 \) is not preferable, but might have a naturally explanation as stated in the last section. We could investigate other influences on the change-density to figure this out. There is not any strong evidence supporting the null hypothesis.

Conclusion

We reject the null hypothesis on the basis of the correlation coefficient and the significance of the regression analysis.

5.3 Discussion and Summary of the results

To summarize the result, we made a table with the research questions and the outcome of the analyse, see table 5.11.

<table>
<thead>
<tr>
<th>Quality focus</th>
<th>ID</th>
<th>Research question</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defects</td>
<td>RQ1</td>
<td>How does the defect-density in reusable components evolve over time?</td>
<td>H10 is rejected. There is 95% probability that the defect-density is different in the releases.</td>
</tr>
<tr>
<td>Stability</td>
<td>RQ2</td>
<td>How does the number of changes per component size in reusable components evolve over time?</td>
<td>We can not reject H02, at a 5% significance level.</td>
</tr>
<tr>
<td>Stability vs. Defects</td>
<td>RQ3</td>
<td>Are the reusable components with high change rate also the most defect prone components?</td>
<td>H30 is rejected. There is a relation between change-density and defect-density for the components.</td>
</tr>
</tbody>
</table>

In answering RQ1, we counted defects for different releases, and divided this on the component size, to get the defect-density. We used a statistical test, called ANOVA to check whether there was a significant difference in the mean-value between the releases. This would give us evidence to conclude if the defect-density changed between the timeperiods. The test made us reject H01 at a 0.05 significance level. We could therefore conclude that there is a difference in defect-density for JEFcomponents in different releases. In addition, the average defect-density decreases with time. The combination of this and the result from test, back us in concluding that the defect-density in the components decrease over the releases in our study.

Ostrand and Weyuker have done a study of faults in a system at AT&T [41], which was embrodied in section 2.8.1. They also observed that new files had higher fault-density than older files. Our study, indicate the same tendency but at a component level. New components, or the most recent modified components, have lower defect density than measured in the older versions. Few studies have put focus upon the evolution of defect-density in reusable components across releases earlier. Therefore, we are in lack of a baseline to compare our results to.

In answering RQ2, we counted changes for different releases, and divided this on the component size, to get the change-density. The procedure was exact the same as
with RQ1. The result of the ANOVA test in this case, did not give us any evidence to reject the null hypothesis. We could not state anything according to the stability progression over several releases. An outlier in the dataset was identified, namely the change-density of JDataaccess in Release 3.1 with value 11,1940. This is a considerably higher value than the other calculations. We have looked at coverage reports of the components in different releases and found that the JEFdataaccess component was not added or modified from Release minor to Release 3.0. This explains the lack of reported changes in this period. We assume that a new release cover an ampler subset of requirements, or changes, than the previous release. For the system to meet the requirements, tasks must be identified. The tasks involve modification of code in components. Which components that are related to the requirements, might be a coincident. Perhaps some components have more basic functionality than others and is therefor less exposed for changes. This should be further investigated. A study with a larger dataset, to eliminate this problems, is also of high interest. There are few empirical studies on this topic. Consequently, we are in lack of a baseline to compare our results on this research question too.

RQ3 relate RQ1 and RQ2 to each other. We wanted to see if there was a connection between the components with high change-density and high defect-density. To do this we calculated the average defect-density and change-density threw the hole time-periode. We used regression analysis to see if there was a linear dependency between the variables. The outcome gave a good implication of this, that let us reject H03. We therefore concluded that there is a relationship between change-density and defect-density for the components under study. We have not examined the cause of this relationship, which could be interesting. A possible line of action is to look at the contents of the change reports and defect reports too see if the code modification in a component introduce defects. Lehman suggests an approach, which resemble a lot like our research regarding RQ3. He suggest this as an approach for establishing baselines to identify feedback mechanisms and control them.

5.4 Validity of results

This section will discuss the threats to validity of our empirical study.

5.4.1 Conclusion validity

A threat to the conclusion validity of the ANOVA testing could be the assumptions about a normally distributed dataset. The test is quite robust, but we did not test the fitting of the distribution. Another threat is how the defects and changes are reported at Statoil. It is necessary that the developers have the same understanding of which category an enhancement should belong to. We know that small enhancements are considered as faults and bigger enhancements as changes, but we do not know the basis of this decisions.

Another threat is the small size of our dataset, this might lead to conclusions with a not sufficient basis. If we had more components and releases to compare, the conclusions would be more reliable. Components should be picked at random to eliminate sources leading to error. This was not possible with the available dataset, because we had to use data from all the components to get sufficient data. As new releases are edited, they could be included in our results to see if they support the same tendency as we discovered.
5.4.2 Internal validity

Missing an inconsistent data is a threat to internal validity. The trouble reporting system at Statoil does not force the reporter to fill out all the fields. Therefore there are missing data in some elements of the reports. We have tried to avoid this with using attributes that are well covered. All report entries have a state field, which shall contain the current state of the defect at report time. We have found some inconsistencies with the name of the state registered at identification time. We have assumed that some state names have the same meaning, but we cannot be 100% certain.

Another threat to internal validity is the ambiguity about the direction of causal influence. This is in the first place a problem associated with RQ3, where we explored the relationship between change-density and defect-density. We conclude that a component with many changes would cause more defects, but the causal influence might be in the other direction, with many defects leading to more changes.

The object of our study is a single group, one project or system only. The focus is on reusable components, but we do not know if this is the factor that caused the results and that the results would be the same if the components were not reusable. In this case it would be preferable to have a control group, a group of comparable components in the same domain that were not reusable. Then we could have seen if the results were addicted to the factor of reusability.

5.4.3 Construct validity

We wanted to investigate the stability of the components over time, and choose change-density as an indication to this quality attribute. In other studies they have measured stability in a different way, but we have also got support from the literature that this is a possible indication. [6]

Mono-operation bias means that the experiment is under-representing the construct, and does not give a whole picture of the theory. This might occur in our research, because there is just one case under study.

5.4.4 External validity

A threat to external validity for all our hypothesis is that all data come from one company and one system inside of the company. With this in mind we can not generalize our results to other systems inside the company, and we can not generalize the results to other companies. The study do not have particularly good external validity. More data samples from comparable contexts are needed to generalize the results. Such studies should consider a random sample of reusable components from multiple projects in various software development firms.

5.5 Evaluation of the change reporting in Statoil

The study has sought to analyse the quality of the reusable JEF components in the SJEF architecture for Statoil. In the process of formulating our research questions we had some problems with getting data from Statoil with the required metrics. We would therefore evaluate the change reporting system at Statoil to see if the system has attributes to measure their own overall goals, and also see what could be added to improve our study.

The article [46] show results from studying different defect classification schemes.
With knowledge gathered from the studies quality views and measurement goals are connected with defect attributes. In this section we will look at the defect reporting in Statoil, and compare the scheme with knowledge from the article.

The general goals defined by Statoil are [69]:

1. Provide project progress information that will demonstrate whether cost, time-scale and quality are being met.

2. To show where improvements in software process and productivity can be made.

3. Demonstrate that customer quality and requirements targets are being met.

The 1st goal is according to the article about evaluating the quality from a value based view. The cost of correcting a defect should be evaluated against the probable customer value. The number of defects and criticality of defects can be used in measuring the impact of reused and investments in software. Unsolved defects present work to be done and can predict the time needed. Attributes with metrics for cost and effort are recommended to monitor the values in a project. Statoil have a metric on the estimated time to resolve a problem in their schemes. The registration of the fields are not impressive, because there are many blanks.

The 2nd goal is according to the article about evaluating software process quality. Defect data can in this case be used in identifying when most defects are injected. The efficiency in which the defects are identified and removed could be measured with proper metrics. Attribute with how and when an defect is defected, would help answering effectiveness of test-activities, to see how effective the activity is to bring the activity to the surface. Registering the cause of a defect, can help to identify product or process problems and perform defect causal analysis. The reports at Statoil have a headline field, a free textual description, this make it difficult to distinguish between types av defects and catagorize the causes. Making the granularity more fine and classifying causes of defects could help Statoil easier find improvement areas. In an incremental development process, the history of modules and components is an important factor in predicting their reliability.

The 3rd goal is according to the article about evaluating product quality from the user’s view. It suggest that the number of problems or defects associated with a software product have impact on the software quality, specially the reliability. An evaluation to see if the customer quality are being met, is to have attributes like customer impact and value, this would be valuable for assigning severity an priority to defects. Statoil do not use this attribute, and the choise must be made on a cost/benefit basis, to see if the information could give a valuable effect.

The article [46] presents three requirements for a good measurement system, identified by Chillarge. We are going to investigate whether the report scheme in Statoil meet this requirements.

Orthogonality to ensure that factors can be evaluated without confounding the effects on the response. The factors must have uniquely identified metrics. In the development project of JEF, there have been some misunderstandings regarding the classification of changes. The developers may have had different comprehension of this conceptions.

Consistency regarding all stages of software development and verification. This means that the same scheme can be used in all phases. The SJEF project and the projects that utilize the SJEF architecture all use ClerQuest as defect and change report system in alle stages of the process.
Uniform across products.

The SJEF project and the projects that utilize the SJEF architecture all use ClerQuest as defect and change report system. The attribute fields in the scheme are not mandatory, leading to blank fields. Different attributes can remain blank in various products.

The article [46] also state that making the value space small for each attribute can make classification easier and less error prone.

5.5.1 Problems related to measure the reuse impact of quality

Tracing the cause of a problem would have a great value for evaluating the impact of reuse. The subsystem that caused a problem could not always be found, and the subsystem attribute are then denoted General. This could reduce the internal validity, because the information is not precise enough. Many of the incident reports and trouble reports have missing fields. A proposal for the reporting process is to prevent closing a report before sufficient information is filled out. The estimate attribute have to many missing fields to make an analyse, and interesting questions around cost and time-scale can not be answered.

5.5.2 Conclusion

An advantage of using ClearQuest as a report system is that the attribute fields can be customized for the project at hand. The scheme Statoil is using could be considered more throughoughly regarding the choice of attributes, metrics and registration processes. This could lead to better monitoring and evaluation opportunities of their goals.
Chapter 6

Conclusion and further work

This empirical study has analyzed defect and change reports in the purpose of assessing the software quality and evolution of reusable components developed by the JEF Team at Statoil ASA in Trondheim. This chapter presents the conclusion and further work of this study.

6.1 Conclusion

On the basis of statistical evidence, we have concluded that the defect-density of the JEF components change over several releases. The average defect-density of the releases indicate that the defect-density decreases as time passes and new versions are released, but we could not provide statistical evidence of this.

We did not arrive at a conclusion on whether change-density change over several releases. The cause of a change might be explained by other factors than the software process only.

The results did not give us any indication of how the change-density evolve with time. Therefore, we could not draw any conclusion about the stability of the JEF components over releases. A change might be caused by other factors than the software product itself. Such causes may be information from the software process’ environment that encourage or oppose software changes.

Furthermore, we got strong evidence that there is a relationship between the change-density and defect-density of the JEF components. The linear dependency between the variables show that components with high change-density also have high defect-density. However, we are not sure whether high defect-density is the cause or effect of change-density.

Two essential threats to the validity of our study are that our data consists of non-random samples of components, and that the reports are retrieved from one single project. Random sampling of a well-defined population is a condition for formal generalization. Our results can only be generalized to new releases of the studied JEF components, and other products developed in the same context.

We conclude that the defect-density of the JEF components evolves, and probably declines, with the number of releases. Our results also provide us with evidence that there is a linear dependency between the defect-density and change-density in the JEF components.
6.2 Further work

Few studies have put focus upon the evolution of defect-density and change-density in reusable components across releases before. Therefore, we are in lack of a baseline to compare our results to. We encourage the initiation of an empirical study of other cases of release-based development of reusable components. Such a study should consider a random sample of reusable components from a well-defined population of projects in various software development companies. A comparison of our results with the results of such a study, could provide a falsification or further generalization of our results.

Furthermore, we couldn’t reject our null hypothesis on change-density. Further empirical studies that focus upon the evolution of reusable components’ change-density is needed to reject this null hypothesis, or potentially verify that it can not be rejected. There is also a need for a study of the cause-effect relationship between defect-density and change-density to assess which of the two that is the cause of the other.

The JEF Team can use the results of this empirical study to establish baseline and make a prediction model. This model can be used for maintaining the studied components, or in planning and managing of the development of future releases of the studied components, or products developed by the JEF Team. If such an approach is chosen, further measurements and studies should be done periodically to update the baseline and the prediction model.
Bibliography


[40] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. Technical report, Ericsson (Grimstad, Norway) and Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU, Norway) and Simula Research Laboratory (Norway), 2004.


66


[65] Trondheim Statoil ASA at Rotvoll. Internal PowerPoint-presentation at Statoil.


