NTNU
Norwegian University of
Science and Technology

Software Defect Analysis

An Empirical Study of Causes and Costs in the Information Technology Industry

Jan Maximilian Winther Kristiansen

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Reidar Conradi, IDI
Co-supervisor: Jingyue Li, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Company X has introduced a classification scheme based on the orthogonal classification scheme by IBM. Analysis of the data in this classification scheme was the primary focus of the specialization project during the fall of 2009. It showed there were significant differences between correction costs of different defect types.

In this study, we want to investigate the root causes for why defects take extensive effort to correct as little research have been carried out within this area. In addition, we want investigate if different project types faces the same challenges with regard to defects which require extensive effort to correct.

Assignment given: 15. January 2010
Supervisor: Reidar Conradi, IDI

PREFACE

This thesis is the result of the subject *TDT4900 - Computer and Information Science, Master Thesis* at the Department of Computer and Information Science, under Faculty of Information Technology, Mathematics and Electrical Engineering, at the Norwegian University of Technology and Science.

I would like to thank my main supervisor professor Reidar Conradi for guidance and support throughout the writing of this thesis, giving me the opportunity to work on the EVISOFT-project from fall 2009 to spring 2010, and giving me the freedom to explore my areas of interest. It has been an exciting year for me. I would also like to thank my co-supervisor researcher and post doc Jingyue Li for helpful suggestions. He has given me consistent and good advice throughout my work on this thesis. I would like to thank professor Tor Stålhane for giving me constructive advice regarding my choice of problem to research, and for valuable insight into other organisations. I would like to thank the representatives from Company X for allowing me to work on this project, for giving me access to their defect tracking database, and for the two summer internships I have had at Company X.

Last, I would like to thank my mom, Irene Winther, and dad, sivilingeniør Jan Gunnar Kristiansen for continuous support throughout my studies, and proofreading my thesis.

Trondheim, June 30, 2010.

Jan Maximilian Winther Kristiansen

EXECUTIVE SUMMARY

The area of software defects is not thoroughly studied in current research, even though it is estimated to be one of the most expensive topics in industries. Hence, certain researchers characterise the lack of research as a scandal within software engineering. Little research has been performed in investigating the root causes of defects, even thought we have classification schemes which aims to classify the what, where and why regarding software defects. We want to investigate the root causes of software defects through both qualitative and quantitative methods.

We collected defect reports from three different types of projects in the defect tracking system of Company X. The first project was a project concerned with development of a general core of functionality which other projects could use. The second was a project aim at the mass-software market, while the third project was tailored software to a the needs of a client. These defect reports were analysed by both qualitative and quantitative methods. The qualitative methods were based on grounded theory. The methods tried to establish a theory of why some defect require extensive effort to correct through analysis of the discussions in the defect reports. The quantitative methods were used to describe differences between defects which required extensive or little effort to correct.

In the qualitative analysis, we found four main root causes which explain why a group of defects require extensive effort to correct: hard to determine the location of the defect, long discussion or clarification of the defect, incorrect corrections introduces new defects, and implementation of missing functionality or re-implementation of existing functionality. A comparison between the four root causes and project types revealed the root causes were influenced by the project types. The first project had a larger degree of discussion and incorrect corrections than the second and third projects. The second and third projects were more concerned with hard to locate defects and implementation of missing functionality or re-implementation of functionality. Similarly, a comparison against another organisation showed there were differences with regard to root causes for extensive effort. This showed how systematic analysis of defect reports can yield software

process improvement opportunities.

In the quantitative analysis, we found differences among extensive or little effort to correct defects and project types. The extensive to correct defects of the first project were due to incorrect algorithms or methods, injected during the design phase, and high risk of regressions. In the second project, the extensive effort to correct defects were due to algorithms, methods, functions, classes and objects, were concerned with the core, platform, and user interface layers and injected during the design phase, and lower regression risks. In the third project, the defects which required extensive effort to correct were due to assignation and initialisation of variables, or function, classes and objects, related to the core-layer, injected during the coding phase, and average regression risk of medium. The little effort to correct defects in the core project were concerned with assignation or initialisation of variables, checking statements, lower regression risk, injected during the code phase. In the second project, easy to correct defects were concerned with checking statements in the code which had a low regression risk. In the third project, defects which required little effort to correct were due to checking statements, interfaces with third party libraries, lower regression risk and stem from requirements. The quantitative analysis contained high levels of unspecified values for little effort to correct defect. The levels of unspecified attributes were lower for defects which required extensive effort to correct.

We concluded there were differences among project types with regard to root causes for defects, and that there were differences similar between different levels of effort required to correct defects. However, the study were not able to measure how these differences influenced the root causes as the study was performed in a descriptive manner.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

ADCT Analysis, Design, Coding and Testing

B2B Business to Business

B2C Business to Consumer

CMM Capability Maturity Model

COTS Commercial Off The Shelf

EVISOFT EVidence based Improvement of SOFTware engineering

IEEE Institute of Electrical and Electronics Engineers

ISO International Organization of Standardization

NTNU Norwegian University of Technology and Science

ODC Orthogonal Defect Classification

SPI Software Process Improvement

TDD Test-Driven Development

USD United States Dollar

CHAPTER 1

INTRODUCTION

The chapter gives an introduction of why this research is important to pursue, and states the context of the project. Last, the chapter will give an overview of the thesis structure.

1.1 Motivation

Software engineering topics have been under continuous research since well before the first conference on software engineering in 1968 [Naur and Randell, 1969]. However, little attention has been paid to research of correction of software defects. Correction of software defects are the process of detecting, locating, and correcting defects in software[1] [Society, 1990]. Approximately 20% of all software defects take 80% of all the required effort to analyse, isolate and correct software defects [Boehm and Basili, 2001]. Authors like Lieberman characterise the overlook of research in this particular area as a scandal within software engineering [Lieberman, 1997].

Software development life cycles, software quality, software testing, software maintenance and software defects are related to and affect each other by complex relationships. Lyu states the purpose of any software engineering activity is to prevent any defects from being introduced in the first place [Lyu, 2007]. However, it is impossible to guarantee a defect free software system [Lyu, 2007]. Software defects is estimated to cost U.S. industries USD 60 billion every year [Tassey, 2002]. In contrast, the entire U.S. software sales market is estimated to be worth approximately USD 180 billion [Tassey, 2002]. Consequently, Jones characterise poor software quality to be the most expensive challenge humanity currently is facing [Jones, 2008a].

Most of the research performed within the area of software defects is how much costs poor software quality imposes. Other aspects are how much costs software

[1] We use the term "defect correction" for the term debugging in to be consistent as explained in Section 2.4.

testing and software maintenance impose on projects, and estimation techniques for these costs. However, little research has been devoted to investigate why certain software defects are hard to correct, or how much effort they require. Defect tracking systems are common in software development organisations today; hundreds of thousands of defect reports are stored in these systems. Defect classification schemes are designed to a answer how, what and where about software defects [Freimut, 2001]. However, none of the most implemented defect classifications schemes have any attributes for classifying the effort required to correct defects [Freimut, 2001].

The motivation of this study is to continue from the work performed during the fall of 2009. The study showed there were a difference between software defects which required extensive effort to correct and those who did not require much effort to correct [Kristiansen, 2009]. Defects which required extensive effort to correct were inserted earlier in the software development life cycle, and the type of corrections was classified as either algorithms, methods, memory management or concurrency management. On the other hand, defects require little effort to correct were the result of errors in functions, classes, objects, or statements. The purpose of this study is to investigate the research questions:

- What are the root causes of the defects which take extensive effort to correct?
- What differences exists between projects in the same organisation with regard to root causes for defects which require extensive effort to correct?

The study is a case study of Company X[2].

1.2 Project Context

The thesis is a continuation of the project performed in *TDT4520 Program and Information Systems, Specialization Project*. The thesis is written as a part of the EVidence based Improvement of SOFTware engineering (EVISOFT) project. EVISOFT is sponsored by the Research Council of Norway. It has 10 industry participants, and is a research partnership between SINTEF, Norwegian University of Technology and Science (NTNU), and University of Oslo (UiO). The purpose of the project is to provide experience-based Software Process Improvement (SPI) which help companies deliver software with high quality, within deadline and within budget [EVI].

[2]We call the company "Company X" throughout this thesis

1.3 Thesis Structure

The thesis is divided into seven chapters and is organised in the following way:

- Chapter 1 serves as a introduction to the thesis.
- Chapter 2 is the result of a literature study performed during the project. It gives an introduction to concepts and state-of-the-art within software engineering, software quality, software maintenance, software defects and software testing.
- Chapter 3 describes the practices as Company X relevant to this study.
- Chapter 4 details the research process and techniques used in this study. Research questions is stated, and work plan is presented.
- Chapter 5 presents the results of the performed study.
- Chapter 6 discusses the results described in Chapter 5, and provides an evaluation of the validity of the results.
- Chapter 7 brings the thesis to a close with concluding remarks, recommendations, and directions for further research.

The following appendices are attached to this thesis:

- Appendix A contains the draft of research paper *Cost Drivers of Software Corrective Maintenance: An Empirical Study in Two Companies* authored by Li, Conradi, Stålhane and Kristiansen (the candidate). This research paper has been submitted to the International Conference on Software Maintenance (ICSM) 2010. It was accepted at June 29th 2010 for presentation at ICSM 2010 and to be published in the conference proceedings of ICSM 2010.
- Appendix B contains draft of the research paper *Enhancing Software Defect Tracking System to Facilitate Continuous Software Quality Assessment and Improvement* authored by Li, Conradi, Stålhane and Kristiansen (the candidate). At the time of writing, this research paper has been submitted to IEEE Software and is currently under review.

CHAPTER 2

STATE OF THE ART

The purpose of this chapter is to establish a theoretical background for the project. The focus of this study will be on software defects and effort spent correcting software defects. However, it is necessary to explore research areas which influence or touches software defects. Hence, we include the subjects software engineering, software quality, software maintenance, and software testing. Software defects affect each of these research areas. For instance, poor software quality may be manifested through severe software defects, or software maintenance may be costly due to many defects requiring extensive effort to correct. Last, we explore relevant research methods for this study. The following digital sources was consulted: ScienceDirect[1], ACM Digital Library[2], IEEE Xplore[3], and JSTOR[4].

2.1 Software Engineering

The following sections will give an overview of software engineering through the main concerns of current software engineering practices and a look at the software development life cycle.

2.1.1 Background

Software engineering is defined by IEEE standard 610.12-1990 [Society, 1990] as:

> (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1). [Society, 1990, page 67]

[1] http://www.sciencedirect.com
[2] http://portal.acm.com
[3] http://ieeexplore.ieee.org/Xplore/guesthome.jsp
[4] http://www.jstor.org/

The scope of this definition is general. It consists of both the practice and the study of the same field. The contents of the definition can be broken down to activities that are performed during specific points in the software development life cycle. These activities are concept analysis, requirement analysis, design, implementation, testing, integration and maintenance [Braude, 2001]. The following is stated in the introduction to the report from the first conference on software engineering in 1968 held by North Atlantic Treaty Organization (NATO):

> The phrase "software engineering" was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering. [Naur and Randell, 1969, page 8]

This is still the current state of affairs. Mahoney states how software engineering conferences routinely starts their conferences by asking and answering the question "Are we there yet?" [Mahoney, 2004]. We still do not have a branch of engineering established in theoretical disciplines. However, we have developed tools and processes which aid us throughout the software development life cycle. On the other hand, these processes and tools have a weakness that is fundamental in how software engineering is performed. Lyu states this fundamental weakness of software engineering as an engineering discipline in [Lyu, 2007]: the decisions made are based on human judgement and bias rather than laws and required processes.

Software engineering research have been characterised by being problem-driven research in the past [Osterweil, 2007]. This has been of mutual benefit for both the practitioners and researchers. However, Osterweil argues on how software engineering research should also be curiosity-driven to complement the problems faced by the practitioners.

2.1.2 The Software Development Life Cycle

The software development life cycle is a structured approach on how we develop software [Marakas, 2006]. The model which is used for software development may differ from each organisation. There are multiple ways of organising the software development life cycle. However, the legacy of models from the last decades include the waterfall life cycle, the iterative model and the agile model.

The Waterfall Life Cycle

The waterfall life cycle consists of several phases which are performed in a successive order. Each step is completed and perfected before the next step starts. In

orther words, the software development life cycle is assumed to be linear and predictable. A software development life cycle based on the waterfall life cycle may include phases such as requirements analysis, design, implementation, integration, testing and maintenance [Braude, 2001]

Royce has been credited for describing the waterfall life cycle in the paper [Royce, 1987] in 1970. Larman and Basili explains the reasons for the early widespread adoption of the waterfall life cycle in the seventies and eighties [Larman and Basili, 2003]: The waterfall process is simple to learn and use, the people adopting the life cycle thought it was a simple, traceable and measurable process, and it was recommended in literature and software engineering courses.

Critics of the waterfall life cycle like Larman claim [Royce, 1987] was misinterpreted by the industry which lead to the large scale adoption of the model [Larman, 2003]. Larman has assembled comprehensive evidence of project failures due to the project following the waterfall life cycle as the software development life cycle of the project in [Larman, 2003]. A fair share of the criticism the waterfall life cycle has received concerns the assumption of linearity and predictability of processes. The environment where the software is developed in is in constant flux, thus not adapting to the current circumstances might yield software which is not useable in the worst case. Parnas and Clements lists several problems of the waterfall life cycle in their search for a rational software engineering life cycle [Parnas and Clements, 1986]:

1. The customer seldom knows exactly what they want.
2. Many small details do not become apparent before they are required.
3. Human minds are unable to comprehend the wast details required in order to construct error free software.
4. Projects are subject to change due to external conditions.

The waterfall life cycle does not address these issues problems, and in [Larman, 2003] there are numerous examples of how projects failed due to these reasons.

The Agile Life Cycle

The agile life cycle has emerged in the last decade as a reaction to the classical document-driven software development life cycles. However, the principles behind the agile life cycle are not new. The principles stem from iterative and incremental development and these principles have been in use by the industry since the 1950s [Larman and Basili, 2003]. These ideas have been in constant evolution throughout the decades, and have evolved into different approaches. Dingsøyr

and Dybå distinguish these approaches into several main categories [Dybå and Dingsøyr, 2008]: crystal methodologies, dynamic software development methods, feature-driven development, lean software development, scrum, and extreme programming. These methodologies differ in how they organise the software development life cycle and other areas where they give unique attention. For instance scrum focuses on project planning in uncertain environments where development is planned in small steps, while extreme programming focuses on implementing best practices into the life cycle.

The agile life cycle focuses on agility, the ability to adapt to rather than predict real world situations. The word "agility" is defined by Erickson, Lyytinen, and Siau as:

> means to strip away as much of the heaviness, commonly associated with traditional software development methodologies, as possible to promote quick response to changing environments, changes in user requirements, accelerated project deadlines, and the like. [Erickson et al., 2005, page 89]

Erickson, Lyytinen, and Siau argue the traditional software engineering life cycles cannot respond to change quickly enough due to comprehensive processes and inertia. Ågerfolk and Fitzgerald state how the agile software development life cycle is based on experience from software engineering practitioners [Ågerfalk and Fitzgerald, 2006]. These decades of experience advice the use of communication, flexibility, innovation and teamwork in order to succeed with delivering a project on time and within budget. The leading promoters of the agile software development life cycle wrote the "Manifesto for Agile Software Development" in order to promote the core values of the life cycle [Beck et al.]:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responding to change over following a plan.

The manifesto shows the key differences in how agile life cycles distinguish themselves from waterfall inspired life cycles. However, their organisation of the process are similar to a certain degree. The activities performed during the agile life cycle can be broken down to four: analyse, design, code, and test. These four activities form a cycle, which is called a ADCT-cycle. Life cycles based on waterfall or agile are similar because they perform the same activities. However, the difference between waterfall and agile life cycles is in how they perform the activities during the project. Waterfall is a single-pass life cycle throughout a project, while

the agile life cycle performs the activities multiple times in cycles throughout the project.

Agile life cycles are often coupled with test-driven development (TDD). TDD is a test-first approach which can be integrated with the ADCT-cycle. The approach consists of five steps [Beck, 2003]:

1. Create required tests for the desired functionality.
2. Run all tests to verify that the new test fails.
3. Implement desired functionality in the minimal amount of code required.
4. Run all tests to verify that all tests pass.
5. Refactor and clean up the unnecessary code.

2.2 Software Quality

Software quality is standardised through the international standard ISO/IEC 9126-1:2001 [ISO/IEC, 2001]. Quality is defined by [Society, 1990] as:

> (1) The degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations. [Society, 1990, page 60]

This implies that software quality is related to both how software conforms to its specified requirements, and what the user excepts of the software. Garvin looked upon quality from several perspectives based on different demographics [Garvin, 1984]:

- The transcendental view is a philosophical view related to the undefined characteristics of a product.
- The user view relates to how the user perceives the quality of the product.
- The manufacturing view is regarding how the product conforms to specifications, design and process.
- The product view looks upon quality as the features of the product.
- The value based view balances cost versus price of the product

Software quality is important. A business developing software must develop software on time, within budget, and with high quality in order to remain competitive [Slaughter et al., 1998]. Others, like Osterweil, have predicted software quality to

Table 2.1: *The list of main hazards which could affect industries severely because of low software quality [Jones, 2008a].*

Industry	Hazard
Airlines	Safety
Defense	Security
Finance	Financial
Healthcare	Safety
Insurance	Liability
State, local governments	Economy
Manufacturing	Operational
National Government	Records
Public utilities	Safety
Telecommunications	Services

be a more important success criterion in the future [Osterweil, 1996]. Software quality is assured through activities employing techniques, processes and tools throughout the software development life cycle. These quality assurance activities are performed at certain points in the life cycle. For instance, such activities include but is not limited to software testing, software inspections, or code reviews.

Low software quality can impose several hazards on key US industries [Jones, 2008a]. These hazards are listed in Table 2.1. Jones further lists four software quality hazards for all industries:

1. Software is accused for major business problems.
2. Poor software quality is developing to be one of the most expensive topics in the world.
3. Organisational culture regards software personnel as a matter of necessity instead of professionals.
4. Every industry is looking into how to improve software quality.

2.2.1 Costs of Software Quality

The costs of software quality is not well understood. Osterweil estimates that at least 50-60% of the effort in producing software systems are spent in assuring a certain standard of quality [Osterweil, 1996]. The costs implied by assuring software quality can be divided into two types: conformance and non-conformance costs [Slaughter et al., 1998]. Conformance costs are the costs associated with

controlling defects. This is done through preventing defects from occurring, and evaluating or auditing products to assure conformance to specifications and standards. Non-conformance costs are the costs concerned with failures related to internal or external factors.

There are economical arguments for implementing software quality assurance activities in the software process. Juran and Gryna argues that "quality is free". Their argument tells if voluntary conformance costs to prevent defects are increased, the decrease of involuntary non-conformance costs will exceed the increase. The sum of the costs are then positive and thus "quality is free" [Juran, 1999]. On the other hand, others believe attaining a high degree of software quality is not economically feasible as other software development activities would have to be sacrificed in favour of software quality [Slaughter et al., 1998]. The consequence of this view is postponing software quality assurance activities until the later phases of the project. Developers would perform ad-hoc testing at the very end instead of planning and executing tests rigorously throughout the project. Software quality cannot be tested in at the last minute of a project [Ammann and Offutt, 2008].

Licences distributed with software in general states the software is provided "as-is", and software developers takes no responsibility in defects occurring during use unless a contract establishes the responsibilities. This is regarded as a problem for software quality. Schneier thinks the software quality problem will persist until a relationship of responsibility is formed between software developers and consumers [Schneier, 2004].

2.2.2 Software Quality Models

How do we measure quality in software? This has been a central question when developing models which tries to describe software quality. The ISO 9126 describes software quality through a set of characteristics: functionality, reliability, usability, efficiency, maintainability, and portability [ISO/IEC, 2001]. These characteristics are divided into 20 sub-characteristics and these are further divided into indicators. The quality of the software is determined by measuring a indicator with a predetermined metric and comparing it against set targets.

The ISO 9126 standard suffers from several limitations. First, it lacks a rationale for selecting which quality characteristics to select, and the selection process is arbitrary. Second, all the measurement is done at the lowest level possible. It is not possible to measure top level characteristics, thus making the model proposed untestable. This makes it hard to know if the model represents a complete and concise definition of quality. Other models, like the quality model of McCall, suffers from the same problem [Kitchenham and Pfleeger, 1996].

2.3 Software Maintenance

Software maintenance is necessary since all software evolve throughout their life in use [Belady and Lehman, 1976]. "Software maintenance" is defined by [Society, 1990] as:

> the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment. [Society, 1990, page 46]

However, this definition has received criticism due to it considers software maintenance to be a post-delivery activity [Canfora and Cimitile, 2001]. An organisation should plan for and prepare software maintenance activities before delivery.

Lientz and Swanson describe three types of software maintenance [Lientz and Swanson, 1980]: corrective, adaptive, and perfective maintenance. Corrective maintenance is the maintenance performed in order to remove defects in the software. Adaptive maintenance is to adapt the software to the current working environment due to environmental changes. Last, perfective maintenance is to enhance the current software with changes proposed by users.

The costs associated with maintenance of software increases with the age of the software increases [Porter, 1997]. The research on software maintenance have been focused to address the costs through studying how to do modifications, how to live with evolution and how to manage the maintenance process [Porter, 1997]. The costs of software maintenance is influenced by factors. Porter classified these factors into four categories [Porter, 1997]: product, people, process and task. Product factors relate to how system attributes and characteristics affect software maintenance costs, while people is related to how attributes of individuals and groups affect software maintenance. Process factors is how individuals and groups perform the activities affect software maintenance. Last, they propose factors related to tasks. Tasks are how maintenance is performed.

Lehman suggested several laws governing software evolution and maintenance in [Lehman, 1996]:

> An E-type program that is used must be continually adapted else it becomes progressively less satisfactory. [Lehman, 1996, page 108]
>
> As a program is evolved its complexity increases unless work is done to maintain or reduce it. [Lehman, 1996, page 109]
>
> The program evolution process is self regulating with close to normal distribution of measures of product and process attributes. [Lehman,

> 1996, page 109]
>
> The average effective global activity rate on an evolving system is invariant over the product life time. [Lehman, 1996, page 110]
>
> During the active life of an evolving program, the content of successive releases is statistically invariant. [Lehman, 1996, page 110]
>
> Functional content of a program must be continually increased to maintain user satisfaction over its lifetime. [Lehman, 1996, page 110]
>
> E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment. [Lehman, 1996, page 111]
>
> E-type Programming Processes constitute Multi-loop, Multi-level Feedback systems and must be treated as such to be successfully modified or improved. [Lehman, 1996, page 111]

A "E-type system" is a system designed to solve a real world problem [Lehman, 1996]. These laws illustrate how important it is to perform software maintenance throughout a software products life. Failing to maintain software in a production environment might yield software with dire consequences for its users. However, the laws have been controversial and has faced criticism from several researchers when they were proposed [Lehman, 1996].

2.4 Software Defects

It is impossible to produce defect-free software products, however, the main purpose of any software engineering activity is to prevent defects from being introduced in the first place [Lyu, 2007]. A defect[5] is defined by [Society, 1990] as:

> An incorrect step, process, or data definition in a computer program. [Society, 1990, page 32]

There is confusion in the terminology used concerning the terms defects, mistakes, errors and failures. The difference between the terms is explained by [Society, 1990]:

> The fault tolerance discipline distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error). [Society, 1990, page 32]

[5] We use the term defect instead of "fault".

Software defects are unique compared to physical defects in a product. They are harder to assess, and more cunning [Pham, 2003]. Software does not deteriorate like physical products. The deterioration of software is a product of side effects from changes in the product in order to assess defects or changed requirements. This implies the total defect count may increase after release during maintenance. There is a risk of introducing new defects when changing the code.

Software defects can be injected during any phase of the software development life cycle. Jones lists the following as sources of defects: requirement errors, design defects, coding defects, documentation defects and incorrect corrections [Jones, 2008b]. A defect is injected when an employee does a mistake in the work performed which creates a defect. A failure in the software might manifest itself and be discovered in either testing or inspections. However, not all defects are discovered before delivery. Such defects are latent defects and can be harder to locate, and advanced users are more likely to discover these than normal users [Jones, 2008b].

Lyu proposes for techniques which is used in the software development life cycle in order to reduce the amount of defects [Lyu, 1996]:

- Defect prevention takes aim to reduce the number of defects introduced while producing the software in the software development life cycle. This is done indirectly in any software engineering activity [Lyu, 2007].
- Defect removal is to detect defects by software verification or software inspection. The main goal is to eliminate introduced defects. Strategies to achieve this may be dynamic analysis, or formal inspections of code.
- Defect tolerance is to provide continuous software service which satisfies given requirements despite a defect having occurred in the software.
- Defect forecasting is to estimate where new defects are likely to emerge in the software.

It is favourable to attain a high degree of defect removal efficiency in the organisation according to Jones [Jones, 2009]. The effects of this will have ramifications well beyond reducing the number of existing non-discovered defects in the software. The effects will allow the organisation to hit minimum schedules, maximise productivity, increase work environment satisfaction, fewer delivered defects, lower maintenance costs and lower risks of legal issues. The most effective way of improving software productivity is to lower the number of defects in the software [Jones, 2009]. This reduction can happen through defect prevention and defect removal, as described by [Lyu, 1996].

2.4.1 The Behaviour of Software Defects

Many instances have tried to quantify the economic loss due to defects in software. A report published by National Institute of Standards and Technology (NIST) claims software defects costs U.S. industries USD 60 billion every year [Tassey, 2002]. In contrast, the entire U.S. software sales market is estimated to be worth approximately USD 180 billion [Tassey, 2002]. The economic impact of software defects have made software quality an important issue in both industry and academia. In addition to economical costs, one could argue software defects impose negative social effects on both software users and software developers [Lieberman, 1997].

Basili and Boehm assembled a list of ten rules of thumb which main purpose were to highlight pitfalls in software engineering. Seven of these rules are directly related to how defects impact the project [Boehm and Basili, 2001]. First, it is 100 times more expensive to correct a defect after delivery than during the requirements or design phase. Second, 40 to 50 percent of the effort in the project is spent on rework which could have been avoided. Third, 20 percent of the defects results in 80 percent of the rework and 80 percent of the defects come from 20 percent of the modules while half of modules are almost free of defects. Developing high dependability software are often 50 percent more expensive than low dependability software, and 90 percent of the downtime comes from 10 percent of the defects. However, investing the 50 percent extra is well worth it if the software is to be maintained. Last, 40 to 50 percent of software contains defects which are nontrivial. The results described by Boehm and Basili in [Boehm and Basili, 2001] gives an impression of how software defects influence the total costs of software projects.

Westland did a study of a software project in order to describe the economical cost behavior of defects [Westland, 2004]. His results shows that defects only generate significant costs if they required redesign of the system, and the costs growth of delaying a correction of the defect till later phases supports the exponential model described by Boehm and Basili in [Boehm and Basili, 2001]. The costs of correcting a defect decreased during the lifetime of the project as developers became more familiar with the design and requirements of software. The number of defects discovered and reported is related to how long the project is running [Westland, 2004].

Software defects are introduced during phases of the software development life cycle. However, the type of typical defects introduced by each phase is distinct, and they have a varying degree of impact on budgets and schedule [Jones, 2009]. Jones describes the defects introduced during the requirements phase as the hardest

to locate and correct. On the other hand, the defects introduced through the code are the most numerous, however, they are the easiest to locate and correct. The defects introduced during the design phase are the most grave, while defects in the documentation can be severe if ignored. Incorrect corrections of defects are very difficult to locate, and poorly designed test cases are often more of a burden than help. Another aspect is the data quality, which is hard to measure. These arguments underline his message of a good software defect removal process include formal inspections, testing, and static analysis.

Kim and Whitehead studied the defect correction time of two open source systems [Kim and Whitehead, 2006]. The defect correction time in their study is calculated by comparing the date of the revision with the defect and the revision with the correction of the defect in the version control software. They make no distinction between the severities of the defects. 50 percent of the defects required between 100 to 300 days in order to be corrected, and the median correction time was approximately 200 days. This method of estimation is inaccurate. It is unlikely a developer spent 100 days with investigation and correction of a defect.

2.4.2 Software Process Improvement with Software Defects

Grady claims software defect data is the most valuable source of information for software process improvement decisions [Grady, 1996]. Further, the defect data provides a way of comparing improvements done against historic defect data in order to measure the effect of the improvements. He argues how ignoring defect data might yield dire consequences for business performance of an organisation through reduced customer satisfaction and increased operational costs. Further, it may have cultural consequences for the operating organisation ignoring defect data. First, workforce might become comfortable with reactive thinking and fire-fighting, and managers might reward this behaviour. Second, the reactive environment have a motivational effect on workforce through higher levels of stress and uncertainty [Grady, 1996].

There are three ways organisations approach the handling of defects according to Basili and Fredericks [Fredericks and Basili, 1998]. The most basic approach is the firefighters who have no established processes for defect management other than the ones required to keep track of them. However, firefighters do not use the defect data to facilitate any change in the software processes. They have defined processes for collection and handling of defect data, but the defect data is never used. The second strategy is to be reactive. Organisation employing a reactive strategy use the collected defect data to improve how they work. The third strategy is being proactive. An organization employing a proactive strategy analyses defect

data continuously in order to prevent similar defects from occurring in the future. They share defect data across the organization in order to elicit areas on where to improve.

There are several factors one should consider when trying to improve processes in an organisation. First, the defect classifications should be repeatable. The same defect should be classified in the same way by two different people. Second, it is important to have well defined goals in order to have focused improvement activities. These goals must be reasonable in order to be achievable. Last, management must support the goals and process improvement incentive [Fredericks and Basili, 1998].

2.4.3 Software Defect Reporting

Reports of software defects can be collected in four ways. Mullally et al. describe these four as interactive user reporting, online user reporting, automated per-incidence reporting, and automated reliability reporting [Li et al., 2008]:

1. Interactive user reporting requires users to initiate the defect reporting process through contacting the software organisation directly. This is typically done by a software support department which fills out a predefined form. The defect is then analysed by technical personnel.

2. Online user reporting allows the user to initiate the defect reporting process by providing feedback on a website. The user fills in a predefined form in order to report the defect.

3. Automated per-incidence reporting reports defects automatically when a defect occurs. The necessary data is collected automatically and sent to the software organisation for analysis.

4. Automated reliability monitoring is a continuous reporting process which periodically communicates all the failures or other critical data which have occurred since the last communication with the software organisation.

The main difference are degree of automation and user involvement during the defect reporting process. The user is the source of the data in interactive user reporting and online user reporting, while the data is collected automatically in the last two. These four approaches were evaluated against a set of criteria: accuracy, correctness, comprehensiveness, normalisation, actionable, effort by software developers and effort by users. Their conclusion favours automated reliability monitoring because it best satisfies the criteria, and provides highest quality of data for the software developers. However, automated per-incidence reporting can be

sufficient if usage data is not relevant.

Bettenburg et al. investigated how developers perceived the quality of the defect reports [Bettenburg et al., 2008]. They did a survey among software developers to identify what information was important to correct the defect, and asked the software developers to rate the quality of selected defect reports. They found a significant mismatch between what information developers considered helpful and what information defect reporters provided. The information developers used most in order to correct defects were steps to reproduce, observed behaviour, expected behaviour, stack traces, test cases and screenshots. Steps to reproduce, observed behaviour, and expected behaviour were the information most frequently provided by defect reporters. However, stack traces, test cases and screen shots were seldom provided by defect reporters. Defect reporters preferred to report information such as product, version, and operating system instead which were not considered very helpful to correct a defect by software developers.

2.4.4 Software Defect Classification

It is important to classify defects as they contain information regarding the quality of processes and products. The information gathered from defects can be used to track and control progress of projects, and improve the project processes [Freimut, 2001]. This is done capturing the what, why and how about defects. The scope of the defect classification scheme is to cover aspects of defects such as location, timing, symptom, result of the failure, mechanism, cause, severity and costs [Freimut, 2001].

Defect classification schemes suffers from several problems according to Thibodeau [Thibodeau, 1978]. First, the attributes in the defect classification schemes include ambiguous, non-orthogonal, and incomplete attribute values. Second, there are too many attributes in the defect classification schemes. Last, there is confusion among the root cause of the defect, the symptoms of the defect, and the actual defects. In order to address these problems outlined by [Thibodeau, 1978], Freimut suggests several quality attributes a defect classification scheme should hold [Freimut, 2001]. The attributes in the classification scheme, and their available values, should be orthogonal. The attribute values should be complete such that for every defect there is a value which can be selected. The number of values for each attribute should be kept to a small amount. Last, every attribute value should be described by a textual description. A classification scheme following these guidelines should be easy to use, and improve the accuracy of the data captured [Freimut, 2001]. In addition, several classification schemes impose extra quality attributes, like the Orthogonal Defect Classification scheme developed by

IBM [Chillarege et al., 1992]:

- The classification scheme should provide consistency across phases.
- The classification scheme should provide uniformity across products.

This implies the defect classification scheme should be useable in any phase during the project, and in any project in an organisation.

Freimut compared the three defect classification schemes in use today with regard to their scope in [Freimut, 2001]. All three classification schemes investigated capture information regarding location, symptoms, and mechanism. However, little effort is spent on capturing costs information regarding effort to locate, effort to isolate, and effort to correct the defect in all three of the classification schemes.

2.4.5 Software Defect Correction

We use the term "defect correction" instead of "debug". Defect correction is defined by [Society, 1990] as:

> To detect, locate, and correct faults in a computer program. [Society, 1990, page 25]

Defect correction still remains as a activity based on trial and error despite the advances in software engineering the 30 years [Lieberman, 1997]. Lieberman notes how the majority of the technique of choice of programmers when correcting defects are inserting print statements in the code. Eisenstadt collected and analysed stories from developers regarding correction of defects which were considered by the developers themselves to be hard to correct [Eisenstadt, 1997]. He classified the stories against three dimensions in his analysis: why the defect was difficult to correct, how the defect was found, and what was the root cause of the defect. First, his findings showed that the correction was difficult due to inconsistencies between the root cause and the symptom of the defect, or defects where the correction process could not use tools. Second, the main techniques for locating the defects were data gathering such as print statements or hand simulation of the code. These were the techniques used in over 80 percent of the stories analysed. Last, the dominant root causes for defects were in third party software or hardware, and defects in code dealing with memory usage.

The findings of Eisenstadt is in line with our findings in [Kristiansen, 2009]. A majority of the defects which were considered hard to correct where related to algorithms, concurrency or memory. In contrast, the majority of the defects which were considered to be simple to correct were related to user interface issues, or statements regarding checking or initialisation of variables in the source code.

Agans suggests nine rules which form a framework that aids the developer in correcting defects [Agans, 2006].

1. Make sure the software developer understands the system by reading existing documentation.
2. Make the defect manifest itself by provoking the failure in the software.
3. Make sure the software developer is looking at the manifested defect in action, and do not theorise what might have happened.
4. Divide the search space for the location of the defect by disqualifying locations.
5. Change only one thing at a time, in order to compare with previous good results.
6. Leave a trail of actions taken in order to make sure the developer did not overlook anything.
7. The software developer should question his assumptions of the defect if the analysis do not yield any results.
8. The software developer should not hesitate to ask colleagues for insight from other perspectives.
9. The defect is not corrected if the manifestation of the defect vanished without correcting the defect.

However, the rules are not elicited from empirical evidence, but from the authors own perception of analysing and correcting defects.

There has been research on how we can estimate the effort required to correct a defect. Weiss et al. used the search engine Lucene and k-nearest neighbours clustering techniques in order to estimate the effort required to correct the defect, and then compared it to the actual effort for correcting the defect [Weiss et al., 2007]. Their first experiment were off by 20 hours on average from the actual effort, and only 30 percent of the predictions where within a ± 50 percent interval of the actual effort. Their best effort managed to predict effort which was off by on average 7 hours off from the actual effort, and almost 50% of the defects where withing a ± 50 percent of the actual effort. However, higher accuracy decreased the applicability of the technique due to computation needs.

The model developed by Weiss et al. in [Weiss et al., 2007] was the basis of the work of Hassouna and Tahvildari [Hassouna and Tahvildari, 2010]. They proposed four enchantments to the technique: data enrichment, majority voting, adaptive

threshold and binary clustering. In general, every method applied produces improvement over the original work. However, it is required to have a substantial database of historical effort data available in order to estimate efficiently.

Evanco developed a statistical model to estimate the effort required to correct a defect in [Evanco, 1995]. The model was based on characteristics of the defect, such as number of components involved and complexity of the software architecture. Their model was calibrated against the effort data from 5 different projects, and the effort required re-estimated with the same data. Their model underestimated the effort required to isolate the defect, and overestimated the effort required to correct the defect. However, their work is meant as a basis for further work, and it is unknown how their model performs against other data sets. A newer study by Evanco presented in [Evanco, 2001] found that variables affect the effort required to isolate or correct a defect differently. The variables for defect locality had a positive correlation to the effort required to isolate or correct the defect. In other words, a defect discovered during unit testing requires less effort to isolate and correct than a defect discovered during systems testing due to defect locality.

Ramanujan, Scamell and Shah studied how individual, software, and organisational characteristics affects software maintenance effort [Ramanujan et al., 2000]. They used a program which gathered the time every participant used to perform certain maintenance tasks in source code. A survey was performed in order to assess the experiences of participants in software development. They found that experience levels of professional practitioners impacts the effort required to correct defects. Novice software developers should according to the study not be assigned to complex software maintenance tasks. In order to reduce effort, they recommend the software developer organisation to use coding conventions and structured development approaches. Another approach they suggested to reduce maintenance effort is to improve project planning by introducing realistic deadlines and time pressure.

Lucia, Pompella and Stefanucci presented an estimation model for corrective maintenance using multivariate least squares regression [Lucia et al., 2005]. They based their model on five metrics: the number of maintenance task requiring modification of source code, the number of tasks requiring correction of data misalignment, number of miscellaneous maintenance tasks, the total number of tasks, and the size of the system in thousands of lines of code (kLOC). They found that the size of the component in kLOC to be maintained have a larger impact on effort required than the size of the changes. This is in-line with the study of Niessink and Viet [Niessink and van Vliet, 1998]. Their best performing model had one prediction which was off by over 25% of the actual effort and the average error was $\pm 25\%$. They argue this is a significant result due to the nature of software maintenance

tasks; their extent is not known at the start of a project.

Ahn, Suh, Kim and Kim proposed an effort model based on function points [Ahn et al., 2003]. A function point is a metric invented by Albrecht and is a measurement of the functionality provided to a user [Symons, 1988]. The model described by Ahn, Suh, Kim and Kim in [Ahn et al., 2003] took personal, organisational and technical factors called into account. Personal factors accounted for were domain knowledge, programming language proficiency, and knowledge of the software. Technical factors were how modules are structured, module independence, readability of source code, and reusability of legacy software. Environment factors were correctness of documentation, conformity with standards, and testability of the software. Effort in man weeks were then determined to be related to function points through a exponential equation:

$$Effort = 0.054 \times FP^{1.353} \tag{2.1}$$

Their model was tested against one data set, and further testing against new and larger data sets were required in order to draw conclusions. However, they found that the personal, organisational and technical factors had less influence on the effort than expected.

Niessing and Vliet investigated how several factors influenced the maintenance effort [Niessink and van Vliet, 1998]. The factors were grouped into three main factors which measured the amount of change which affects flow-of-control, the increase of system size and the amount of changes in modules. They found that tasks which change the flow-of-control in the software had the most influential impact on effort, followed by increase of system size. They discovered factors might influence factors for analysis, coding, and test differently. Therefore, one should carefully consider the process which is used for elicitation of factors which may influence efforts. They further emphasised the importance of following a standardised process in order to measure the elicited factors more precisely.

Shukla and Misra constructed a neural network in order to estimate maintenance effort [Shukla and Misra, 2008]. They used 14 technical factors which could be grouped into complexity, size, and structured programming concepts. However, they did not include social and organisational factors which influence the defect correction process [Aranda and Venolia, 2009]. The model of Shukla and Misra [2008] managed to estimate the effort required with a mean magnitude of relative error of 5.1% to 5.8%. However, their model was not cross validated against other data sets.

Slaughter and Banker investigate how software development practices affected software maintenance effort [Slaughter and Banker, 1996]. They studied two or-

ganisations and used regression techniques in order to analyse the data sets. Their findings gained insight of how influential certain practices may be on the software maintenance effort. The use of code generators had a significant impact on the required software maintenance effort. This was due to how code generators may change the flow throughout the software, thus imposing extra effort on software developers in order to understand the new code when they need to do changes. The use of Off The Shelf (OTS) components decreased the effort required due to changes to 3rd-party components are more likely to be minor than in-house developed components. The use of a structured process or technique decrease the software maintenance effort required. This is in-line with research presented above.

2.5 Verification of Software

The purpose of verification of software is to help developers create software with high quality through discovering defects early in the development cycle [Harrold, 2000]. The following section will elaborate on both dynamic and static software testing techniques and explain the software testing process.

2.5.1 Dynamic and Static Analysis

Verification of software can be done through dynamic analysis or static analysis. Dynamic analysis is a characterisation of techniques which is used to examine software behavior during the execution of code [Harrold, 2000]. These techniques include random, functional, control flow, data flow, mutation, regression, and improvement testing [Juristo et al., 2004]. Static analysis is a collection of techniques which investigate software which is not under execution. Static analysis techniques are inspections [Fagan, 1986], or formal techniques based on logic and mathematics like deductive methods, abstract interpretation or model checking [Dwyer et al., 2007].

Dynamic analysis and static analysis are both essential quality assurance activities. However, there are different advantages and disadvantages with employing either of them [Harrold, 2000]. Dynamic analysis techniques can be automated and executed with ease, allows the software to be tested in its execution environment, and provide confidence the software behaves according to the intention. On the other hand, dynamic analysis cannot be used to show the absence of defects. It can only be used to show the presence of defects. The results obtained from doing dynamic analysis in an particular environment or version of the software cannot be generalised to any environment or another version of the software.

Static analysis techniques based on inspections use the creativity of the human mind to detect defects. However, humans are error-prone, and like dynamic analysis techniques, cannot be used to show the absence of defects [Osterweil, 1996]. On the other hand, software inspections has proven to provide substantial cost savings for organisations. IBM, Motorola, NASA, and Allianz report that up to 95 percent of detected defects before the testing phase are found by software inspections [Rombach et al., 2008]. Hence, the ramifications of software inspections include shorter delivery time and lower development costs [Rombach et al., 2008]. The techniques based on logic and mathematics are abstractions of the software. Thus, the main limitation is the precision of the model which is used. Consequently, some static analysis techniques can prove the absence of defects [Osterweil, 1996].

2.5.2 Software Testing Process

Testing is in general performed late in the project during the waterfall life cycle, or continuously during the agile life cycle. Graham et al. describes a test process independent from the underlying software development life cycle [Graham et al., 2008]:

1. Planning and control.
2. Analysis and design.
3. Implementation and execution.
4. Evaluation of exit criteria and reporting.
5. Test closure.

It is important the testing process is planned and adapted to the needs of the organisation [Graham et al., 2008]. Hence, organisations should plan their testing based on their own policies, strategies and exit criteria. An important artifact is the test plan. The test plan is a document which contains information regarding schedule, approach, resources, and activities [Society, 2008]. The policies and strategies of the organisations is used to design tests which runs in a test environment and has a certain set of test conditions [Graham et al., 2008]. The test cases are created and executed during the implantation and execution phase based on the designs, environments, and conditions created in analysis and design. The test results from the execution are evaluated against the exit criteria and summarised in reports distributed to all stakeholders involved with the testing process. The test closure involves archiving of test cases, maintenance and summarise lessons learnt during the testing process [Graham et al., 2008].

The V-model describes how software testing can be performed in parallel with other development activities. In the waterfall life cycle, software testing is performed at the very end of the project. The V-model aims to involve testers as soon as possible in the software development life cycle. The model introduces the concept of test levels. A test level is a of test activities that shares management and organisation, and is linked to responsibilities [Graham et al., 2008].

- Component testing: the testing of the components functionality.
- Integration testing: the testing of functionality of interfaces between software components.
- System testing: the testing of the system as a whole.
- Acceptance testing: the testing of whether the system is acceptable for the customer with regard to functional and non-functional requirements.

The testing process can be described from a general perspective through seven questions [Bertolino, 2007]:

- Why: The aim is to determine the test objective.
- How: The problem is to do a test selection that matches the system that is to be tested.
- How much: How to determine when the testing objective is reached.
- What: At what levels should the system be tested?
- Where: In what environment will the tests be executed?
- When: When will the test executions occur?

2.6 Relevant Research Methods

There are two main research strategies: quantitative and qualitative research [Ringdal, 2009]. A qualitative research strategy is based on interpretation of textual data. The research strategy is inductive, as we observe phenomena and create a theory of what is occurring. Studies performed with a qualitative strategy observes the subjects in close proximity in their natural environments. The analysis techniques used are informal.

On the other hand, quantitative research strategies are concerned with numbers. The research strategy is deductive. We test theories against phenomena. The study is performed with distance to the subjects in artificial environments. The analysis techniques which is used are formal statistical methods.

One can combine both quantitative and qualitative strategies in a mixed-strategy design. This is called triangulation [Ringdal, 2009]. Oates describes six different ways triangulation of research can be achieved in [Oates, 2006]: strategy, design, time, space, investigator, and theoretical triangulation. The idea is to use multiple sources or choices in either, for instance, the time of data collection or multiple researchers researching the same topic and comparing results, in order to eliminate threats to the validity of the study.

Research design is a plan designed to help the researcher how to collect, analyse, and conclude. Its main purpose is to help the researcher answer the research questions [Yin, 2003]. Ringdal lists five types of research design: experiments, cross-sectional, longitudinal, case study, and comparative design [Ringdal, 2009]. The experiment is the classical example of cause and effect analysis between independent and dependent variables. The cross-sectional and longitudinal designs are based on a time line. Cross-sectional is based on one point of the time line, while longitudinal are based on at least two points on the time line. Case studies and comparative designs are based on the study of units. A unit may be individuals, organisations, countries or anything. Their difference is that case study designs takes one unit into account, while a comparative designs compare multiple units into account.

Yin lists three conditions that should be considered when selecting a research strategy and design [Yin, 2003]:

1. The type of research questions.
2. The level of control the researcher wants.
3. The focus of the study.

2.7 Summary

The literature study has covered a broad perspective. This is due to the lack of research on the reasons for and effort required in order to correct software defects specifically. This has led certain people into calling it a scandal due to the overlook of this particular research area [Lieberman, 1997]. The research performed within this area is focused on effort estimation. However, there are exceptions. Eisenstadt studied “war stories” from software developers against software defects [Eisenstadt, 1997]. He described how defects which required extensive effort to correct were due to inconsistencies between the root cause and the symptoms of the defect, or the current circumstances which did not allow any use of tools in order to analyse the defect.

Several studies explored in this literature study were concerned with estimation of corrective maintenance effort. However, these models were built on selected software metrics from a theory developed by the researchers rather than factors identified from empirical data. We want to identify these root causes which influence the effort to analyse, isolate and correct defects in Company X based on empirical data from defect reports. This leads us to the first question we want to investigate:

- What are the root causes of the defects which take extensive effort to correct?

Company X have different project types aimed at different markets. We also want to look at how the project type influences these root causes. This leads us to the development of the second question we want to investigate:

- What differences exists between projects in the same organisation with regard to root causes for defects which require extensive effort to correct?

CHAPTER 3

CONTEXT OF COMPANY X

Company X is a world leading software development company within their field of operation. The company currently have over 700 employees worldwide where a roughly 400 of them are software developers and quality assurance personnel. Their product line consists of software for the mass-market and software which is customised to the needs of customers. Company X usually have a major release ready for the mass-market every half year. This chapter will give a overview of how they organise their projects, and description of their defect management processes. The chapter is based on the own experiences of the candidate from his two internships at Company X during the summer of 2008 and 2009, the defect tracking system, and the presentation slides created in cooperation with Li and Conradi in Appendix C.

3.1 Background

The following section is based on the presentation held during the meeting with Company X at March 23rd 2010. Company X have been a official part of the EVISOFT-project since October 2007.

1. Company X wants to increase their effectiveness and efficiency of their software testing practices.

2. Company X wants to increase productivity by software reuse in software development and software testing. Software reuse is the process of developing software systems by use of existing software rather than starting from scratch each time [Krueger, 1992].

3. Company X wants to increase the accuracy of estimates for project and release management.

4. Company X wants to participate in experience sharing across organisations.

The study which is part of this thesis is goal one. We need to know why defects fail in order to select the best strategies for improving software testing practices. The first improvement was the result of a gap analysis performed in through the period January to May 2008. The results of the gap analysis showed three areas of interest for improvement: avoid solutions which is prone to risks, reporting defects in a formal way, and prioritise defect management. A customised version of the orthogonal defect classification scheme was introduced to assess the results from the gap analysis. The classification scheme was introduced during May 2009. Currently, there are more than 2000 defects classified with the classification scheme, and over 80 projects have used it to some extent.

3.2 Projects

Company X have different project types and they are distinguished by what part of the product line they affect and what market they are developed against. The following sections will give an overview of the projects, their structure, and the projects elicited for this study.

3.2.1 Project Structure

Company X produces mass market software which is available on almost any computer operating system or handheld device. The projects can be classified in three categories: business-to-business (B2B), business-to-consumer (B2C), and internal projects which are both B2B or B2C. B2B projects are projects from clients which want services or software tailored to their platform, while B2C projects are software and services which are aimed for the mass market. The category which is both B2B and B2C are internal projects which implement the underlying functionality which is common in software and services provided in both B2B and B2C products as depicted in Figure 3.1. Every project have their own designated area in the defect tracking system described in Section 3.3.

3.2.2 Projects Under Study

The study in this report is based on three different projects: Project A, Project B, and Project C. Project A develops software which provides all functionality for other software projects of Company X. The project has been spanning for over a decade, and is able to run on almost any platform. There are currently over 19500 defect reports reported throughout its life. There are 70 and 222 defect reports classified with an effort to fix as “time consuming” and “quick fix” as of 15th of February 2010.

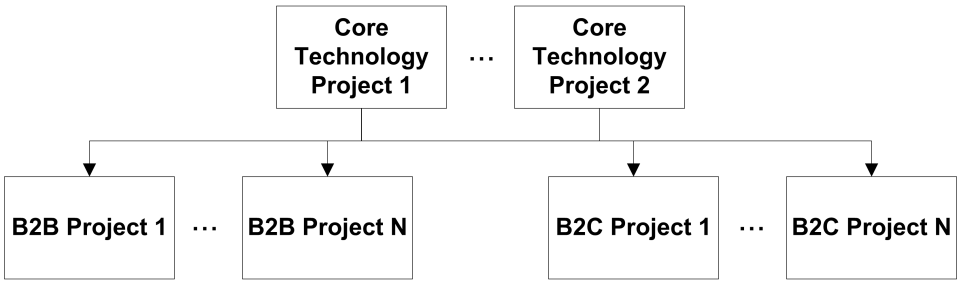


Figure 3.1: *The diagram illustrates the project structure of Company X. The core-projects are used by both B2C- and B2B-projects. Project A is selected from the core technology projects, Project B is selected from the B2C-projects, and the Project C is selected from the B2B-projects.*

Project B is a B2C-project and their flag-ship software product. The project is The project faces stiff competition from the market and users expect high quality. The software product is available on all mainstream platforms such as Windows, Mac OSX, and Linux. The mass-market software is under continuous development, and it faces unique challenges with defect management compared to B2B-projects. The project has been continuously running for over 15 years, and over 262600 defect reports have been reported since 1999. It is important to note that a defect reports is a report of failure, which is a manifestation of a defect, Multiple defect reports could have been reported for the same failure, and the failure could be invalid. For instance, 20 percent of the defect reports reported over a year period from 20th of November 2008 to 20th of November 2009 had an actual correction of the defect. 80 percent of the defect reports turned out to be duplicates of other defects or deemed invalid as a defect [Kristiansen, 2009]. In a worst case scenario, approximately 10 percent of all lines of could would have been affected with a defect if the system size were 3 million lines of code.

Project C is a group of B2B-projects. The data from this group is collected from multiple projects due to the limited time line and number of defects classified with ODC-attributes for each project. Another reason is due to legal concerns and how clients value confidentiality. B2B-projects is done in cooperation with clients which want a customised version of the software for their use. Their clients value confidentiality.

We will refer to the three project types as the core-project (Project A), the B2C-project (Project B), and the B2B-project (Project C) throughout the rest of this thesis.

3.3 Defect Tracking System

Defect tracking is a essential project management tool in Company X. Every project tracks defect reports, feature requests, and change requests. The defect tracking system is based on Atlassian JIRA, which is a purchasable general purpose project management system. The defect tracking system offers functionality to track defects, track projects, custom work flows, reports, analysis, and plug-in system[1].

3.3.1 Defect Reports

A defect report consists of several attributes which aim to describe the defect and aid the developer in correcting it. The section will give an overview of the work flow the defect reports goes through throughout the life cycle of the defect report. However, it is important to distinguish between defect reports and defects. A defect report is a report of the manifest of a defect which became visible to the user as a failure. The same defect could in theory have multiple manifestations yielding different failures, hence implying multiple defect reports may be duplicates.

3.3.2 Defect Report Work Flow

The defect report work flow has been specified by an internal document [Department]. The work flow consists of six states from where the defect is discovered till it is corrected and accepted as a resolution to the defect. The six states are: *NEW*, *EXAMINED*, *CONFIRMED*, *RESOLVED*, *VERIFIED*, and *ACCEPTED*. The states and the possible transitions between them are depicted in Figure 3.2. The defect report is classified as *NEW* when it is submitted into the system. An analysis of the defect report by a quality assurance responsible moves the defect report to the *EXAMINED* state. When necessary, further analysis may be required and when the defect report is found to be a cause of a failure, and it is reproducible, it may be moved to the *CONFIRMED* state. The defect report is moved to *RESOLVED* when a solution is determined for the defect. The correction of the defect is then verified by a quality assurance responsible. If the correction is deemed to be correct, the state of the defect report is changed to *VERIFIED*. The defect is transferred to the accepted state if the client approves the solutions of the defect.

The defect work flow may under some circumstances not be linear as depicted by Figure 3.2. If a new defect report contains enough information to propose a correction of the defect, it may be directly moved from *NEW* or *EXAMINED* to

[1] `http://www.atlassian.com/software/jira/`

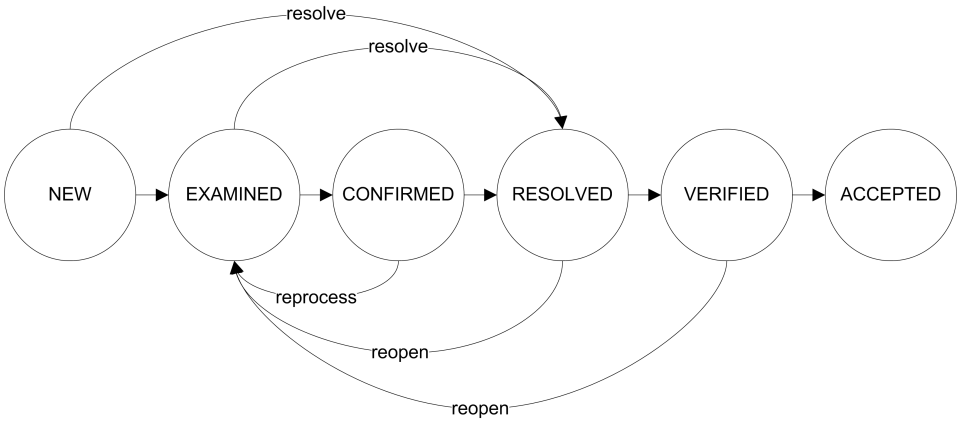


Figure 3.2: *A state diagram of the work flow each defect report goes through from submission (NEW) to completion (ACCEPTED) and how they may move between the states. The work f low acts like a state machine and defines the legal transitions between the states. This is a standard work flow for must defect tracking. The figure is adapted from [Department].*

RESOLVED. A defect may be *reprocessed* if it is in the *CONFIRMED* state and the current information is considered to be incorrect. The defect is then moved from *CONFIRMED* to *EXAMINED* for a new analysis. A defect may be *reopened* if the correction of the defect is deemed to be incorrect. This may happen in the *RESOLVED* or *VERIFIED* state. The state is then changed from *RESOLVED* or *VERIFIED* to *EXAMINED*.

3.3.3 Defect Report Attributes

The defect reports have attributes which contain either qualitative or quantitative information. The following sections will give a brief overview of the attributes of the defect reports

Qualitative Attributes

The defect reports have several qualitative attributes where a defect reporter may fill in information. These attributes are the most important attributes of the defect reports and they give essential information regarding the defect. Typical uses of these fields are to provide descriptions of the defect or discussions among involved parties which may help to correct the defect. These attributes are the *title*, *description*, *environment* and *comments* attributes. They are explained in Table 3.1.

Table 3.1: *The defect reports in the defect tracking system have qualitative attributes. They are listed in this table with a description of what information they seek to capture.*

Attribute	Description
comments	A list of comments from involved parties regarding the defect.
description	A description of the defect.
environment	What environment the software was running in when the defect occurred.
labels	A list of keywords used to group defects together.
title	A short summary describing the defect.

Quantitative Attributes

The quantitative attributes consists of two sets: the standard set of attributes used by Atlassian Jira, and the custom attributes added due to the introduction of the ODC-scheme. The standard attributes and their possible values are listed in Table 3.2. The ODC attributes used in Company X were proposed by Li and Gan in [Li and Gan, 2009]. These attributes and their possible values are explained in Table 3.3.

Table 3.2: *Description of the attributes available for any submitted defect report and all the possible categories one is able to select for each attribute.*

Attribute	Description	Categories
key	The unique identifier of a bug.	A string on the form <ProjectCode>-<DefectReportNumber>
type	The type of the issue	Bug, Change Request, Feature Request, Patch, and Task
status	The current state in the workflow	New, Examined, Confirmed, Resolved, Verified, and Accepted
resolution	The chosen resolution for the bug.	Unresolved, Fixed, Won't Fix, Sitepatch, Implemented, Rejected, Published, Won't Publish, Duplicate, Invalid, and Works for Me
priority	The priority of the bug.	P1, P2, P3, P4, P5, and -
customer priority	The priority specified by the customer.	P1, P2, P3, P4, P5, and -
assignees	The person who is responsible for the bug.	A string with the name of the assignees
reporter	The person who reported the bug.	A string with the name of the reporter
watchers	People that watch for any changes to the bug.	A list with the names of people who watch the issue.
creation date	Time stamp of the creation date of the bug.	Date stamp
updated date	Time stamp from the last change to the bug.	Date stamp
due date	Time stamp of when the bug is due to be finished.	Date stamp
build number	Which build is affected of the defect report.	Number.
components	Which project specific component that is affected by the bug.	A list of project specific components.
affected versions	Which versions of the product that is affected by the bug.	A list of project specific versions.
fix versions	Which version of the product the bug is planned to be fixed.	A list of project specific versions.
severity	How big impact the bug has on the product.	Crashes my computer, Crashes the software, Freezes the software, Other severe, Privacy issue, Security issue, Significant, Site compatibility, Spec violation, and Trivial.
module	The module of the product which is affected by the bug.	A list of project specific modules.
CC	People which is participating in the discussion around the bug.	A list of names of employees.
OS	Which operating the product is running on.	A list of operating systems and versions of operating systems which is affected by the defect.

Continued on next page.

Attribute	Description	Categories
platform	Which platform the product is running on.	A list of platforms which is affected by the defect.
testcase url	URL to any testcases associated with the bug.	A HTTP URL
attached files	Any attached file to the bug report.	A HTTP URL
custwait	If the customer is involved in the bug report, and await a response from the customer.	Yes or No

Table 3.3: *Description of the ODC attributes available for submitted defect reports and all the possible categories one is able to select for each ODC attribute.*

Attribute	Description	Categories
Effort to Fix	The effort used to fix the defect.	Quick Fix and Time Consuming.
Placement – Layer	The layer where the defect was injected.	Website, Core, Mantle, Platform, UI and Device.
Type of Fix	The type of fix for the defect.	Algorithm, Other, Assign/Init, Software Interfaces, Hardware Interfaces, Buffer/Memory Management, Checking, Func/Class/Obj, Timing/Serial and Standard Compliance
Root Cause – Cause	The activity where the defect were injected.	Information, Requirement, Code, National Languages, Design and Build
Root Cause – Identity	The reason why a defect occurred.	Incorrect, Irrelevant and Missing
Severity	The severity of the defect.	Crashes my computer, Other severe, Spec violation, Crashes the Software, Site compatibility, Blocks testing, Security issue, Freezes the Software, Significant and Trivial.
Module(s)	The module which was affected by the defect.	A list of project specific modules
Component(s)	The component which was affected by the defect.	A list project specific components
Regression Risk	The risk of a regression occuring because of the change	High Risk, Medium Risk, Low Risk and No Risk
Effort to Reproduce	The effort spent to reproduce the defect.	Quick Fix and Time Consuming.
Activity to Discover	The activity which was performed when the defect was discovered	stress testing, Not specified, performance testing, Ad hoc browsing, Other, security regression testing, URL toplist browsing and browser tasks testing, fix verification, extensive regression testing, UI/feature specific testing, baseline testing, Operator/OEM testing, General regression, system integration testing, Smoke and release testing

Continued on next page.

Field	Description	Categories
Activity to Reproduce	The activity which must be performed to reproduce the defect	Test basic function/feature, Other, 3rd-party software change, Interaction with environment, Device configuration change, Test scenario with special sequence of operations, Workload/stress testing, and Analysis of unexpected use cases or exception paths

CHAPTER 4

RESEARCH DESIGN

The purpose of this chapter is to detail the research design used in this study. We will describe the research questions, an overview of case studies, and detailed research design for the study. In the end, we describe general threats to validity of this study.

4.1 Research Questions

The following statement were developed at the start of the project:

> I am studying defect correction effort, because I am trying to find out why some defects take substantial effort to correct in order to see how defect management affects software maintenance effort.

The statement were further refined and developed into research questions. The following research questions will be addressed in our research:

- **RQ1:** What are the root causes of the defects which take extensive effort to correct?
- **RQ2:** What differences exists between projects in the same organisation with regard to root causes for defects which require extensive effort to correct?

4.2 Case Study Strategies

The research strategy we chose to use are case studies based on both qualitative and quantitative methods. Yin states that case studies are observation of a contemporary phenomenon in a real world context when the boundaries between the phenomenon and context is not obvious [Yin, 2003]. A research design for a should contain the following components according to [Yin, 2003]: research questions, any propositions, what to be analysed, how the propositions are linked to the data and how to interpret the findings. Yin makes a distinction of case study designs

whether they are holistic or embedded. These two approaches can be used on both single and multiple cases.

A single case design should be used under the following circumstances [Yin, 2003]:

1. When we have a theory and want to test it within certain constraints.

2. When we want to document a special case.

3. When we want to describe a typical situation which may translate to many others.

4. When we want to describe a situation which has been inaccessible to the scientific community.

5. When we want to study the same subject over time (longitudinal).

The multiple-case design does not fit descriptions 1, 2, and 4 above. However, a multiple case design should be focused on replication. The selection of cases to be studied should be chosen based on whether they predict the same results, or they predict contrasting results based on theory [Yin, 2003]. The reason why we chose the case study strategy is because the study satisfies point three, four and five mentioned above. It satisfies point three because software defects are a well-known problem within software engineering, as documented by our literature study. We want to describe reasons and costs of why defects take extensive effort to correct, and compare the results against other organisations. It satisfies point four because Company X is world leading within their market, and a scientific community might show interest in what problems this organisations faces. Last, it satisfies point five as the involvement of the EVISOFT-project is continuous software process improvement work. It is interesting to document the current circumstances in order to facilitate a comparison after improvements have been used to measure the effect of them.

4.3 Research Design

The following section will describe the research design used in this study. The study will perform both a qualitative analysis of defect report discussions, and a quantitative analysis based on the values of the orthogonal defect classifications attributes.

4.3.1 Data Set and Collection

This study will examine defect reports from three different projects: the core project, a B2C-project and a B2B-project. These projects are described in Section 3.2.2. The data will be collected from Company X's software defect database available online. Every defect classified with an effort to correct as "time consuming" or "quick fix" and created before 15th of February 2010 from the three different projects will be collected. The difference between defects which require extensive effort ("time consuming") and little effort ("quick fix") to correct are the effort spent. In the orthogonal classification scheme, effort is recorded on an ordinal scale, and every defect which require equal to or more than 4 hours to correct is classified to be require extensive effort to correct. Similarly, defects which require little effort to correct require less than 4 hours to correct.

4.3.2 Research Design of Quantitative Study

A quantitative data analysis will be performed on the defect attributes collected from the defect reports. The purpose of this analysis is to obtain a general overview of the data set and the differences among the projects. Simple frequency tables will be constructed and summarised.

4.3.3 Research Design of Qualitative Study

A qualitative data analysis will be carried out on the qualitative attributes of the defect reports. The attributes which will be analysed are the "title", "description", and "comments" attributes. The research method will be based on qualitative methods based on coding of the qualitative data [Strauss and Corbin, 1998]. The research design of this part will follow the process proposed by Shannak and Aldhmour [Shannak and Aldhmour, 2009]:

1. Data collection of defect reports from Company X.
2. Data extraction from defect reports.
3. Open coding of the qualitative attributes of each defect report.
4. Generate concepts from the codes.
5. Generate categories with concepts as properties.

4.4 Validity

Wohlin et al. lists the following threats to a study's validity [Wohlin et al., 2000]:

- Conclusion validity which concerns whether the conclusions are the result of a statistical relationships with a given significance.
- Internal validity which concerns whether the relationships discovered between variables are not the result of an unknown variable.
- Construct validity which concerns whether observations are generalisable back to theory.
- External validity which concerns whether the results are generalisable to other populations or contexts.

It has been debated whether the four validity categories could accommodate both qualitative and quantitative studies, since these categories have traditionally been used to validate analysis based on quantitative methods [Whittemore et al., 2001; Trochim and Donnelly, 2006]. Trochim and Donnelly suggests four other categories which are designed to accommodate the threats against a analysis based on qualitative methods [Whittemore et al., 2001]:

- Credibility concerns whether the participants of the study find the results believable.
- Transferability concerns whether the study can be generalised to other contexts.
- Dependability concerns whether changes in the context of the study have been addressed.
- Confirmability concerns the degree where other researchers might come to the same conclusion.

CHAPTER 5

RESULTS

This chapter describes the results from the qualitative and quantitative analysis performed of defect report data from the defect tracking system of Company X.

5.1 General Remarks

In total, 810 defect reports were collected. Every defect report which had the attribute *Effort to Fix* classified as *Time Consuming* or *Quick Fix* and created before February 15th 2010 were collected. 638 of the defect reports had an effort to correct classified as little effort required, while 172 of the defect reports had an effort to correct classified as extensive. They were collected from three project types: core-, B2C-, and B2B-projects. How many defect reports from each project type and what effort to correct they had been classified as, is shown in Figure 5.1. The reason why there were only 14 defect reports collected from B2B-projects is due legal concerns with external clients of Company X. Hence we had restricted access to B2B-project defect reports. The defect reports from the core- and B2C-projects were collected from a single project respectively, while the B2B-project defect reports were collected from two projects.

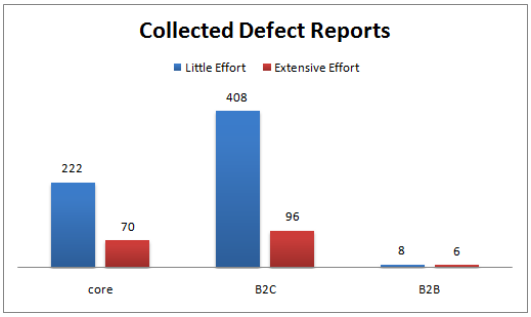


Figure 5.1: *The chart shows the total number of defect reports collected from each project type, and the classified effort to correct.*

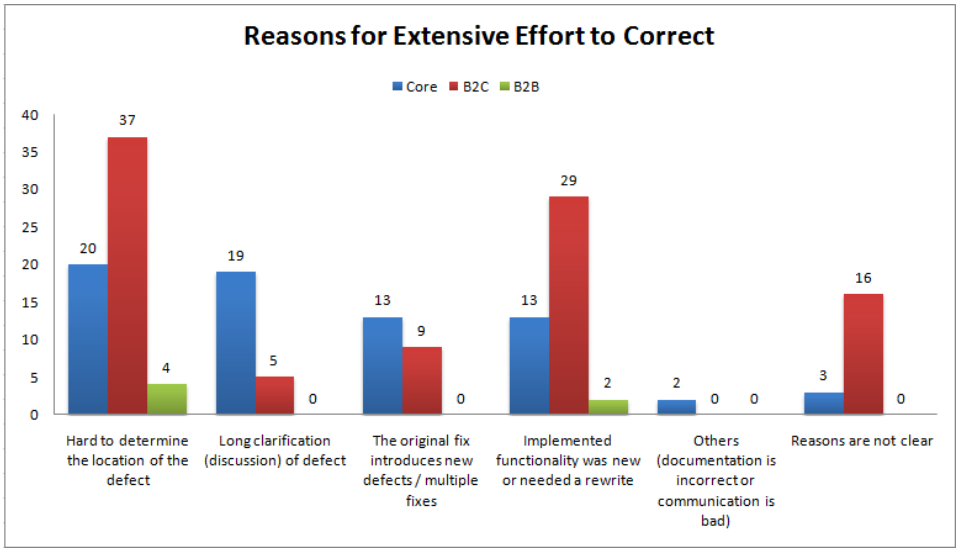


Figure 5.2: *The chart shows the categories and their frequencies for each of the three project types core, B2C and B2B. The y-axis shows the frequency of defect reports, while the x-axis is the categories developed in the qualitative analysis.*

5.2 Qualitative Analysis

The process used to derive the results from the qualitative analysis is described in Section 4.3.3. The results presented in this section are based on 176 defect reports from Project A, Project B, and Project C which had an effort to correct classified as extensive. There were 70 defect reports from Project A, 96 defect reports from Project B, and 6 defect reports from Project C. The results is summarised in Figure 5.2.

The main categories developed from the data suggests there are four reasons for defects requiring extensive effort to correct:

- **It is hard to determine the location of the defect:** This occurred due to inconsistencies between symptoms and the location of the defect. Activities performed to analyse the defect ranged from reading through logs, developing test cases and trying to reproduce the defect.
- **Long clarification and discussion of the defect:** This occurred when the defect Activities which were contributors into facilitating extensive effort were to analyse competitors software for how similar functionality were implemented. Other activities included discussion of requirements specifications, checking whether the defect is a duplication, and technical discussion

of implementation details.

- **The original fix introduces new defects or multiple fixes:** The defect had ripple effects beyond the first correction of the defect. This several cases, multiple corrections of the same defect were required before the defect was corrected. Other corrections of defects introduced new defects which were filed as separate defect reports in the defect tracking system.
- **The implemented functionality was new or needed a rewrite:** The existing functionality did not satisfy the demands other modules required. These defects had often missing or incorrectly implemented functions from the early stages of development. The extensive effort to correct was due to the effort required to re-implement or implement these missing or omitted functions.
- **Other:** The reasons for the defect requiring extensive effort to correct was due to reasons we were to able to connect to any of the other categories. However, the reason is still valid as an explanation of why the defect required extensive effort to correct. The reasons were that documentation was incorrect in one case and the communication was not good enough in the second case.
- **Reasons are not clear:** There were no comments to analyse or the comments did not contain any information which could determined the reason why the defect took extensive effort to correct.

Other Observations

The following observations were noted while reading through the comments of the 172 defect reports.

- **Code Reviews:** Code reviews were performed in the core-project, and were not observed in the other two project types. The review took place when a corrective patch had been developed for the defect, and was conducted by another developer on the same project before committing to the correction.
- **A Black Hole:** The defect tracking system was viewed upon as a black hole of information by a few staff members. That is, information was collected but never used again after the particular defect was corrected.
- **The Persons Involved in Defect Correction:** Early analysis of the defects were frequently conducted by the defect reporter. However, every module and component had a group of persons as owners, which conducted the actual correction of the defect.

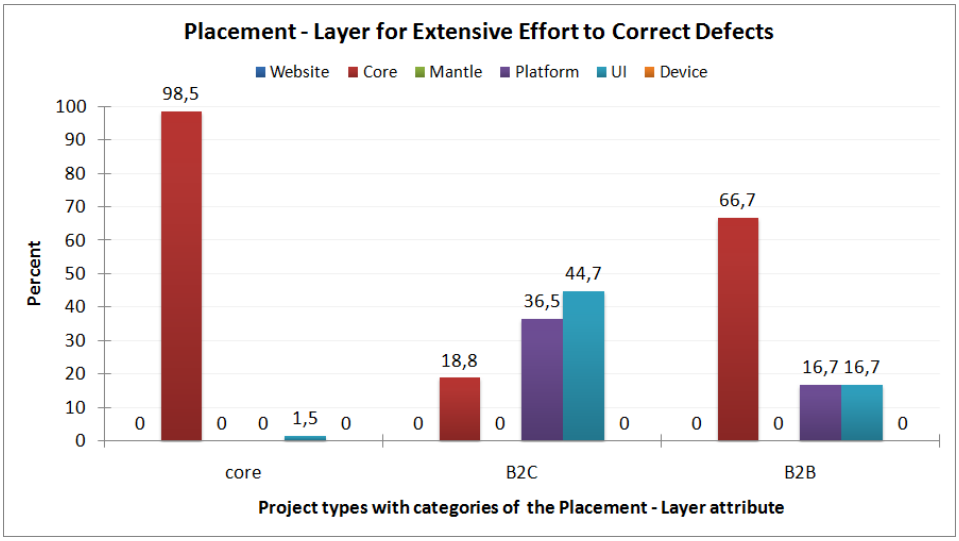


Figure 5.3: *The chart shows the "Placement – Layer" attribute values of the defects which have a "Effort to Fix" classified as extensive effort for each of the three project types in percent.*

5.3 Quantitative Analysis

The attributes *Activity to Discover* and *Activity to Reproduce* were excluded from this analysis due large amounts of unspecified values. Between 95 % and 100 % of both the little effort and extensive effort defect reports had these attributes unspecified. Unspecified values were omitted from the analysis. However, values were recorded to assure completeness.

5.3.1 Comparison of Extensive Effort Defects in the Projects

The following section will list the observations of the comparison between defects which require extensive effort to correct and the differences between project types.

Layer of Placement

The *Placement – Layer* attribute values for each project is shown in Figure 5.3. 98,5 percent of the extensive effort defects related to the core-project were related to the core layer, while 18,8 and 66,7 percent of the B2C and B2B project defects were related to the *core*-layer. On the other hand, 36,5 and 44,7 percent of the B2C-project defects were related to *Platform* and *UI*. Respectively, the percentages for *Platform* and *UI* were 0 and 1,5 percent for core-projects and 16,7 and 16,7 percent

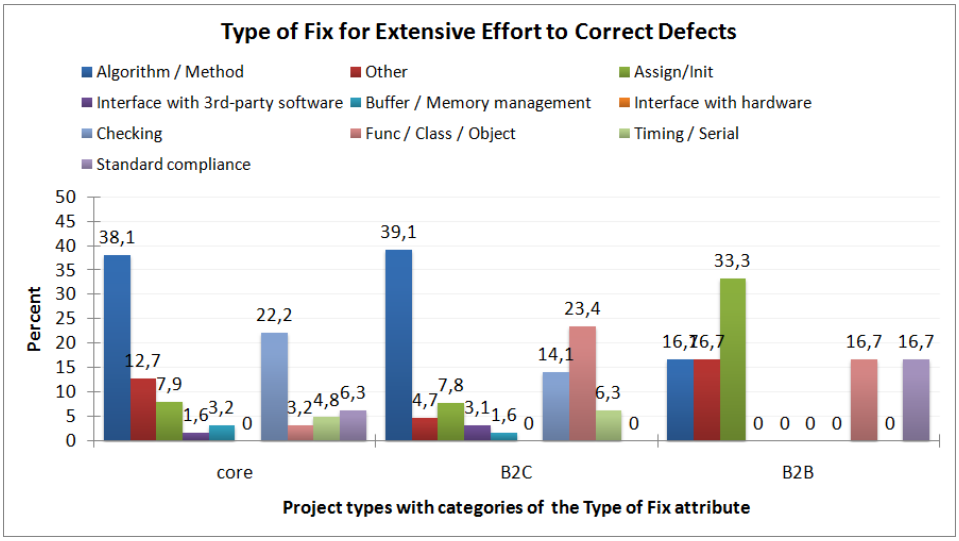


Figure 5.4: *The chart shows the "Type of Fix" attribute values of the defects which have a "Effort to Fix" classified as extensive effort for each of the three project types in percent.*

for B2B-projects. There were unspecified values in both core and B2C. 5,7 and 11,5 percent of the values of *Platform – Layer* were unspecified in core and B2C project defect reports.

Type of Correction

The values for the *Type of Fix* attribute is shown in Figure 5.4. For the core-project, *Algorithm / Method*, *Checking*, *Other* and *Assign / Init* accounted for 38,1, 22,2, 12,7, and 7,9 percent of the defects which required extensive effort to correct. The *Standards Compliance*, *Timing/Serial*, *Func / Class / Object*, *Buffer / Memory Management* and *Interface with 3rd Party Software* categories had less than seven percent each for the core-project and their percentages are shown in Figure 5.4. For the B2C-project, the categories *Algorithm / Method*, *Func / Class / Object*, and *Checking* accounted for 39,1, 23,4, and 14,1 percent of the categories values. Similarly, there were categories which accounted for less than eight percent of the total. These categories were *Assign/Init* with 7,8 percent, *Timing/Serial* with 6,3 percent, *Other* with 4,7 percent, *Interface with 3rd Party Software* with 3,1 percent and *Buffer/Memory Management* with 1,6 percent. The B2B project had 33,3 percent of the defects classified with a *Type of Fix* as *Assign/Init*, while *Algorithm/Method*, *Other*, *Func/Class/Object*, and *Standard Compliance* had 16,7 percent each. There were 10 and 33,3 percent unspecified values for *Type of Fix* in the

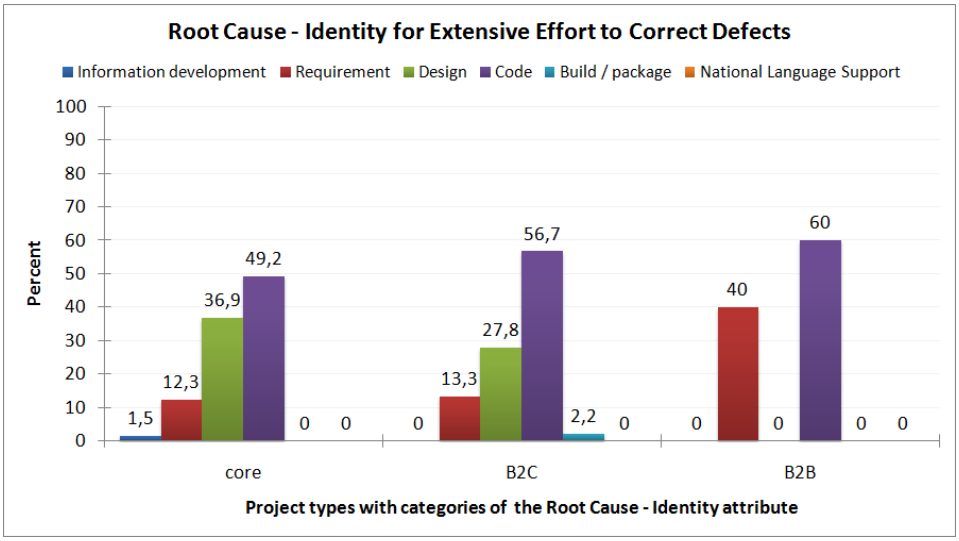


Figure 5.5: *The chart shows the "Root Cause – Identity" attribute values of the defects which have a "Effort to Fix" classified as extensive effort for each of the three project types in percent.*

defect reports from the core and B2C projects respectively.

Identity of Root Cause

For the Root Cause – Identity attribute, the category *Code* had the largest portion in all of the projects. The core-project had 49,2 percent, B2C-project had 56,7 percent and the B2B project had 60 percent classified as *Code*. The next largest portion were classified as *Design*. The extensive effort defects had 36,9 and 27,8 percent from the core and B2C projects classified *Root Cause – Identity* as *Design*. The exception were B2B which had 0 percent of its defects classified as *Design*. On the other hand, the *Requirements* category were the largest among the three project types in B2B with 40 percent. Consequently, 12,3 percent and 13,3 percent of the core and B2C projects defects were classified as *Requirement*. Unspecified values were 7,1 percent for core, 6,3 percent for B2C, and 16,7 percent for B2B. The values for the *Root Cause – Identity* attribute is shown in Figure 5.5.

Cause of Root Cause

The values for the *Root Cause – Cause* attribute is shown in Figure 5.6. The core project had the largest portion of defects classified with a *Root Cause – Cause* as *Incorrect (commission)* with 60,3 percent. The B2C project had similarly 50,6

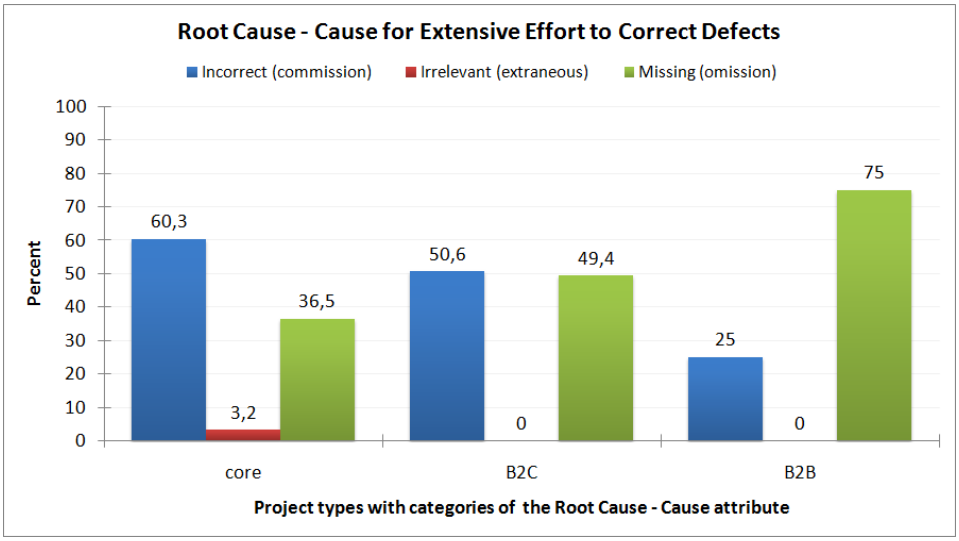


Figure 5.6: *The chart shows the "Root Cause – Cause" attribute values of the defects which have a "Effort to Fix" classified as extensive effort for each of the three project types in percent.*

percent, while the B2B project had 25 percent classified as *Incorrect (comission)*. The category *Missing (omission)* were largest in B2B projects where 75 percent of the defects were classified with an *Root Cause – Cause* as *Missing (omission)*. The percentages for core and B2C projects were 36,5 and 49,4 percent comparably. *Irrelevant (extraneous)* were not used in B2C and B2B projects, while 3,2 percent where classified as *Irrelevant (extraneous)* in the core project. The unspecified values were 10 percent for core, 13,5 percent for B2C, and 33,3 percent for B2B.

Severity

The values for the *Severity* attribute is shown in Figure 5.7. 100 percent of the B2B project's defects had a severity classified as *Significant*, and 54,3 and 60,7 of core and B2C project defects had the same classification. The core project then had 20 percent classified *Severity* as *Crashes the Software*, 14,3 percent as *Spec Violation*, 7,1 percent as *Site Compatibility*, 1,4 as *Blocks Testing*, 1,4 percent as *Trivial* and 1,4 percent as *Other Severe*. In contrast, B2C had 14,6 percent classified as *Crashes the Software*, 13,5 percent as *Other Severe*, 2,1 percent as *Spec Violation*, 1 percent as *Security Issue*, 1 percent as *Freezes the Software* and 1 percent as *Trivial*. All defect reports had the *Severity* attribute specified.

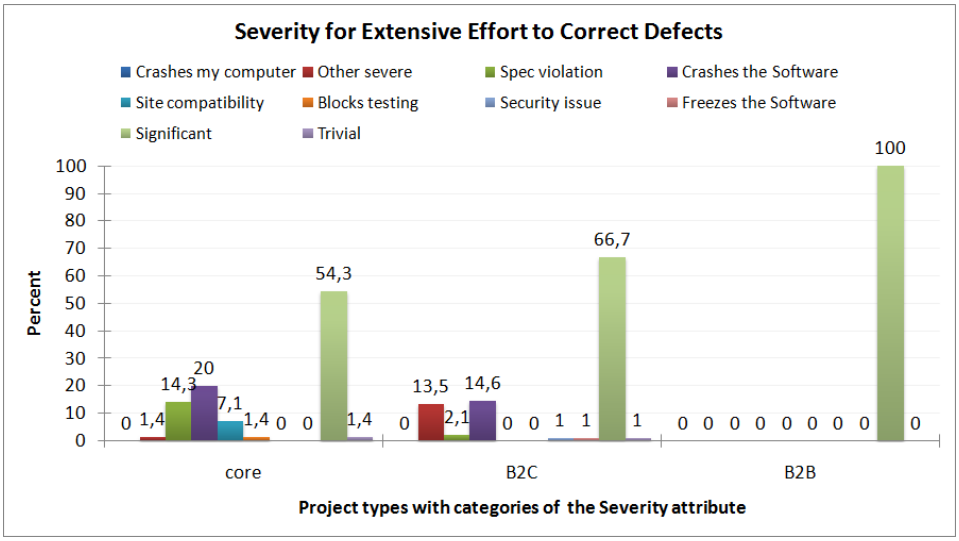


Figure 5.7: *The chart shows the "Severity" attribute values of the defects which have a "Effort to Fix" classified as extensive effort for each of the three project types in percent.*

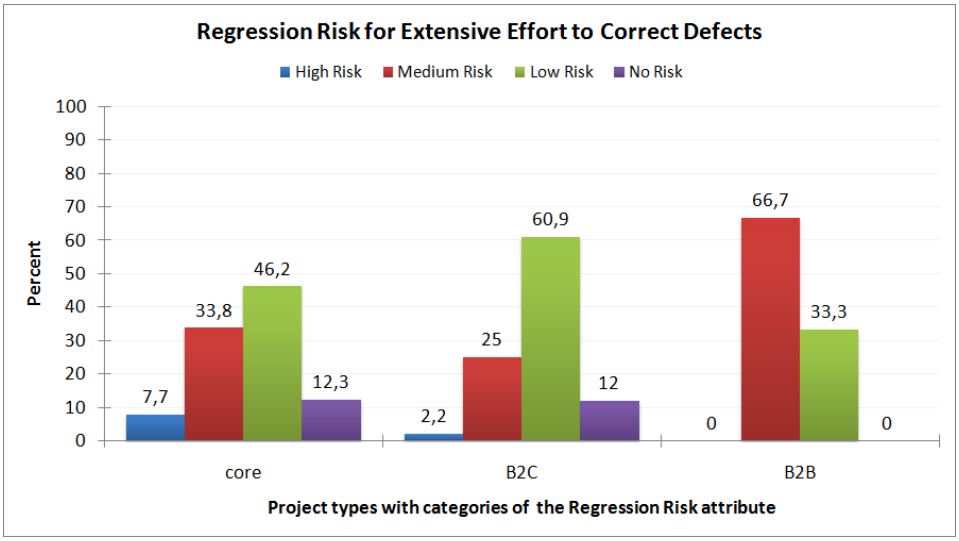


Figure 5.8: *The chart shows the "Regression Risk" attribute values of the defects which have a "Effort to Fix" classified as extensive effort for each of the three project types in percent.*

Regression Risk

The values for the *Regression Risk* attribute is shown in Figure 5.8. The core project have the highest portion of *High Risk* defects with 7.7 percent. Comparably, B2C have 2.2 percent and B2B have 0 percent *High Risk* defects. On the other hand, B2B have the highest portion of *Medium Risk* defects with 66,7 percent. Similarly, the core project have 33,8 percent and the B2C project have 25 percent of the defects classified as *Medium Risk* of regression. Last, the B2C project have the largest portion of *Low Risk* with 60,9 percent, while the core and B2B projects have 46,2 and 33,3 percent each. Both core and B2C had a similar share of 12 percent of *No Risk* regression risks. The unspecified values were 7,1 percent for core project, 4,2 percent for B2C project, and 0 percent for B2B.

5.3.2 Comparison Between Extensive and Little Effort Defects

The results described in the Section 5.3.1 were compared to a data set of defects which required little effort to correct. The results from the comparison each of the ODC attributes are presented below.

Layer of Placement

The values of the *Placement – Layer* attribute is shown in Figure 5.9. There were no large differences in the core project, but the rate of unspecified values went up to 17,1 percent from 5,7 percent and *Core* layer defects decreased to 94,6 percent from 98,5 percent. In the B2C project, there were differences. The amount of defects classified to the *Core* layer decreased to 12,8 percent from 18,8 percent. The portion classified to the *Platform* and *UI* layers changed to 36,1 percent and 50 percent, from 36,5 percent and 44,7 percent respectively. The unspecified values in the B2C project increased to 32,8 percent from 11,5 percent. B2B projects decreased their amount of defects classified to the *Core* layer to 25 percent from 66,7 percent, the *UI* layer to 25 percent from 16,7 percent, and the *Platform* layer increased from 16,7 percent to 50 percent.

Type of Correction

The values of the *Type of Fix* is shown in Figure 5.10. There were several differences discovered. The largest differences were with the *Algorithm / Method* category. *Algorithm / Method* changed in all of the projects to 17 percent, 14,7 percent, and 20 percent from 38,1 percent, 39,1 percent, and 16,7 percent for the core, B2C and B2B projects respectively. For the *Assign / Init* category, the portion increased to 18,8 percent from 7,9 percent for the core project, increased to

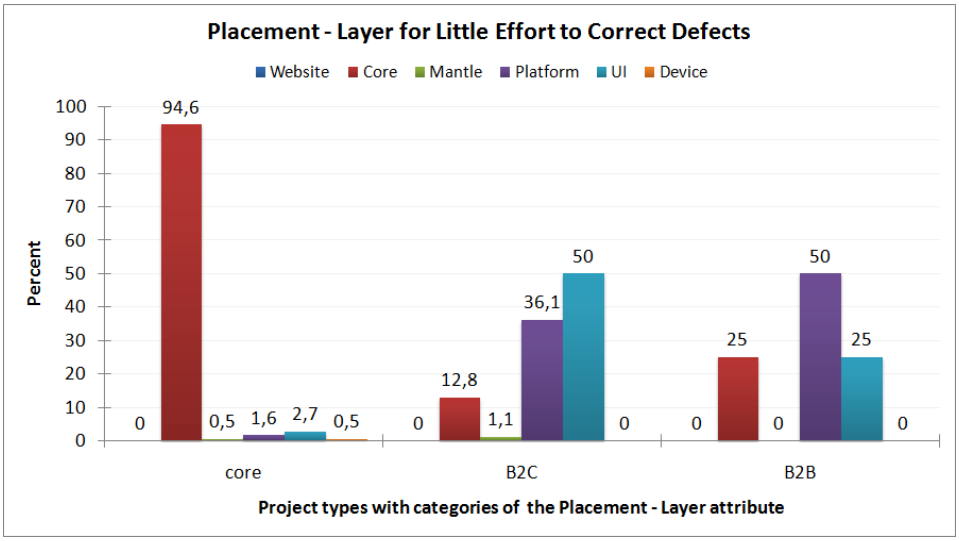


Figure 5.9: *The chart shows the "Placement – Layer" attribute values of the defects which have a "Effort to Fix" classified as little effort for each of the three project types in percent.*

16,3 percent from 7,8 percent for the B2C project, and decreased to 0 percent from 33,3 percent in the B2B project. There were only observed a change in the *Interface with 3rd-party software* category for B2B projects. The portion for B2B project increased to 20 percent from 0 percent. There were changes observed in all projects for the *Checking* category. The values increased to 32,4 percent from 22,2 percent for the core project, increased to 29,3 percent from 14,1 percent for the B2C project, and increased to 20 percent from 0 percent for the B2B project. The category *Func / Class / Object* were unchanged for the core project, but the values increased to 25 percent from 23,4 percent for the B2C project and decreased to 0 percent from 16,7 percent for the B2B project. Minor changes were observed in the *Standard Compliance*, *Timing / Serial*, and *Other* categories. No changes were observed in the *Memory / Buffer Management* and *Interface with hardware* categories. All of the projects experienced an increase in unspecified values to 20,7 percent for the core project, 54,9 percent for the B2C project, and 37,5 percent for the B2B project.

Identity of Root Cause

There were several differences when we compared the identity of root causes among defects which required little or extensive effort to correct. The values of *Root Cause – Identity* for defects which required little effort to correct are shown

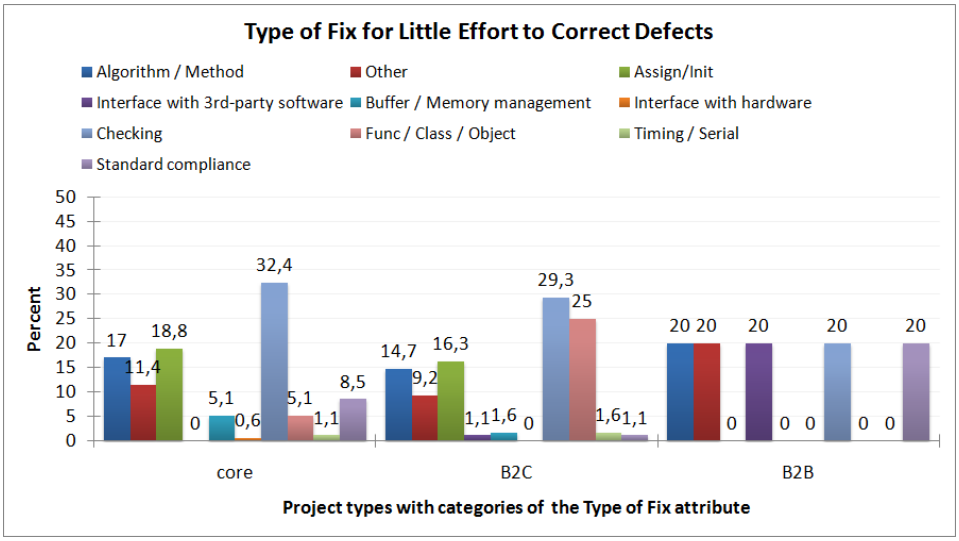


Figure 5.10: *The chart shows the "Type of Fix" attribute values of the defects which have a "Effort to Fix" classified as little effort for each of the three project types in percent.*

in Figure 5.11. There were no changes observed in the *Requirement* category for the core project. However, the value changed to 7,6 percent and 71,4 percent from 13,3 percent and 40 percent for B2C and B2B projects. In contrast, there were no change observed in the *Design* category for the B2B project. The values of the *Design* category decreased to 13,6 percent and 18,1 percent from 36,9 percent and 27,8 percent for the core and B2C projects. Last, there were changes observed in the *Code* category. For the B2C project, the value of the *Code* category increased to 71,9 percent from 56,7 percent. The values of the *Code* category increased to 67,2 percent from 49,2 percent for the core project, while the values decreased to 28,6 percent from 60 percent for the B2B projects. The unspecified values increased compared to the extensive effort to correct defects, and were 20,3 percent, 29,4 percent, and 12,5 percent for the core, B2C, and B2B projects.

Cause of Root Cause

The values of the *Root Cause – Cause* attribute is shown in Figure 5.12. All of the project types saw a decrease in the portion of *Incorrect (commission)* causes. The core project decreased to 54,9 percent from 60,3 percent, while the B2C and B2B projects decreased to 44,5 percent and 0 percent from 50,6 percent and 25 percent respectively. The core and B2C projects experience an increase of *Irrelevant (extraneous)* causes. The increase was to 5,7 percent from 3,2 percent for the core project, and to 3,2 percent from 0 percent for the B2C project. The B2B had

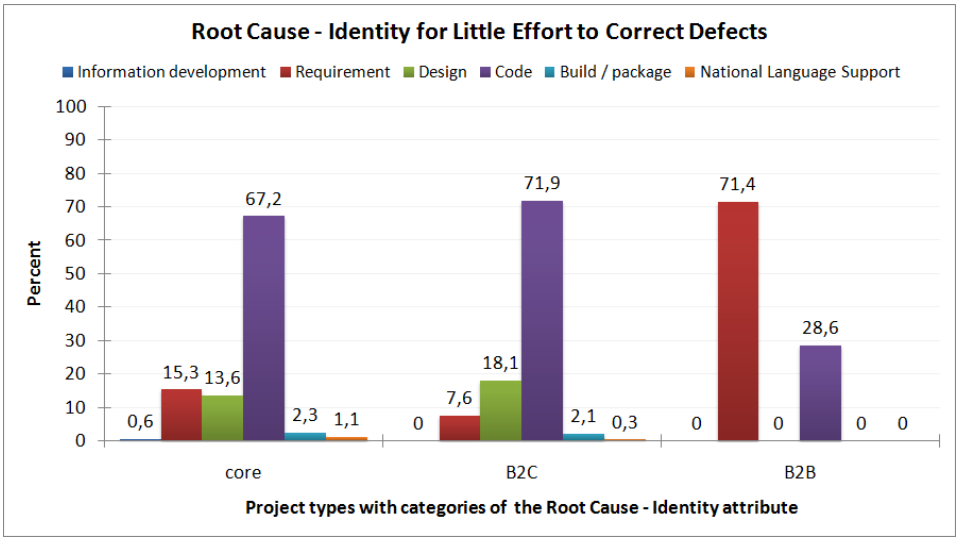


Figure 5.11: *The chart shows the "Root Cause – Identity" attribute values of the defects which have a "Effort to Fix" classified as little effort for each of the three project types in percent.*

no change in the attribute *Irrelevant (extraneous)*. All projects had a a difference in the *Missing (omission)* attribute. The value of *Missing (omission)* increased to 39,4 percent from 36,5 percent for the core project, increased to 52,3 percent from 49,4 percent for the B2C project, and increased to 100 percent from 75 percent for the B2B project. All projects experienced an increase of unspecified values. The unspecified values were 21,2 percent for the core project, 30,6 percent for the B2C project, and 50 percent for the B2B project.

Severity

There were no apparent changes in the dominant category *Significant* in any of the projects. The values shown in Figure 5.13 show a similar distribution as shown in Figure 5.7. The same categories were dominant although minor changes were found. For the core project, *Spec violation* decreased to 11,7 percent from 14,3 percent, *Crashes the Software* decreased to 16,7 percent from 20 percent, and *Site Compatibility* increased to 10,8 percent from 7,1 percent. For the B2C project, *Other Severe* increased to 19,4 from 13,5 percent. No changes were noted for the B2B project.

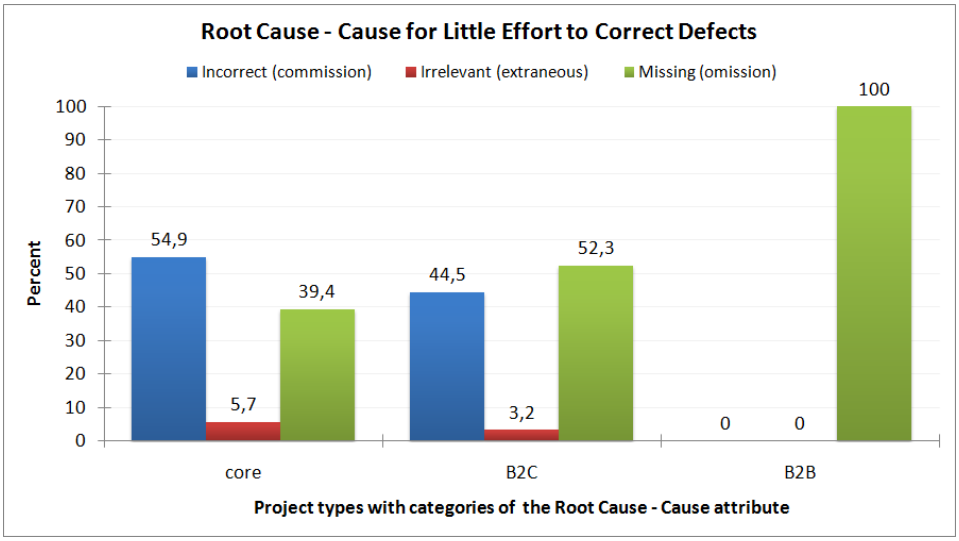


Figure 5.12: *The chart shows the "Root Cause – Cause" attribute values of the defects which have a "Effort to Fix" classified as little effort for each of the three project types in percent.*

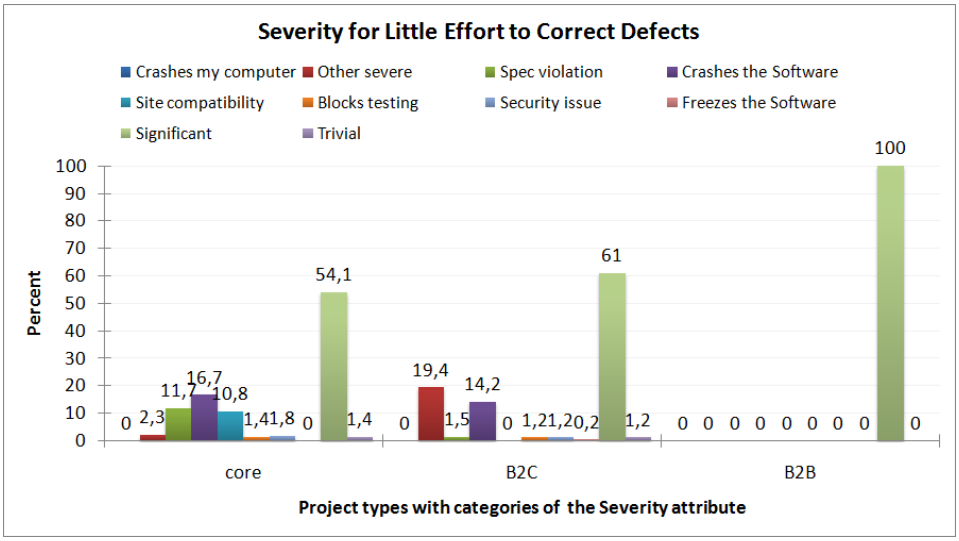


Figure 5.13: *The chart shows the "Severity" attribute values of the defects which have a "Effort to Fix" classified as little effort for each of the three project types in percent.*

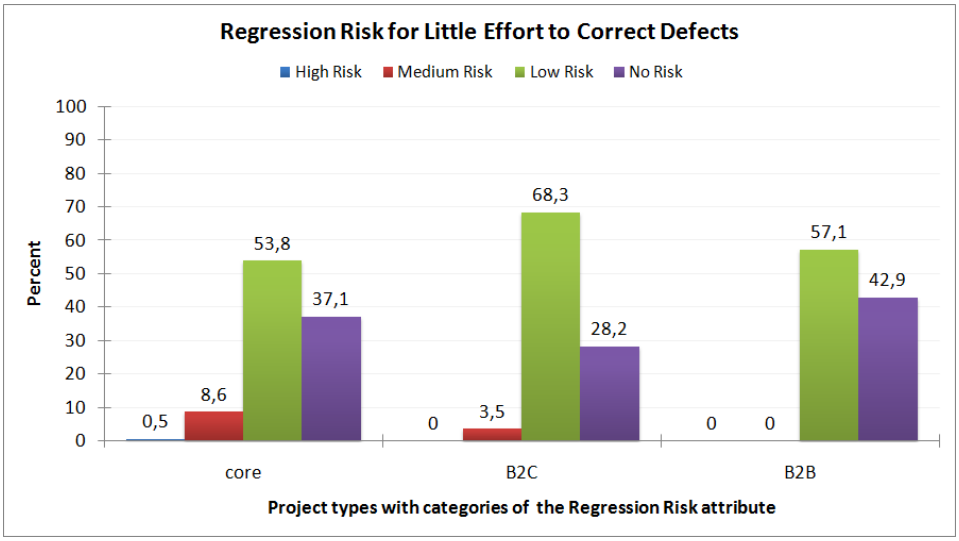


Figure 5.14: *The chart shows the "Regression Risk" attribute values of the defects which have a "Effort to Fix" classified as little effort for each of the three project types in percent.*

Regression Risk

There were noted several changes in the *Regression Risk* attribute values. For the core project, *High Risk* decreased to 0,5 percent from 7,7 percent, *Medium Risk* decreased to 8,6 percent from 33,8 percent, *Low Risk* increased to 53,8 percent from 46,2 percent, *No Risk* increased to 37,1 percent from 12,3 percent, and *Unspecified* increased to 16,2 percent from 7,1 percent. For the B2C project, *High Risk* decreased to 0 percent from 2,2 percent, *Medium Risk* decreased to 3,5 percent from 25 percent, *Low Risk* decreased to 68,3 percent from 60,9 percent, *No Risk* increased to 28,2 percent from 12 percent, and *Unspecified* increased to 29,7 percent from 4,2 percent. For the B2B project, *Medium Risk* decreased to 0 percent from 66,7 percent, *Low Risk* increased to 57,1 percent from 33,3 percent, *No Risk* increased to 42,9 percent from 0 percent, and *Unspecified* increased to 12,5 percent from 0 percent. The values of the *Regression Risk* attribute is shown in Figure 5.14.

CHAPTER 6

DISCUSSION AND EVALUATION

The following chapter will interpret the results of the qualitative and quantitative analysis presented in Chapter 5. The results will be compared versus other research. In the end, a evaluation of the validity of the study is performed.

6.1 Discussion

The first part of this discussion is regarding the results from the qualitative analysis, and the last part is regarding the results from the quantitative analysis.

6.1.1 Results from Qualitative Analysis

The results from the qualitative analysis signalised there were significant differences in root causes with regard to project types. The results described in Section 5.2 shows there were four different root causes for defects taking extensive effort to correct:

- It is hard to determine the location of the defect.
- Long clarification and discussion of the defect.
- The original fix introduces new defects or multiple fixes.
- The implemented functionality was new or needed a rewrite.

These four root causes corresponds to a defect requiring extensive effort to locate, decide, correct, or due to ripple effects. It was evident the defects required extensive effort to correct were due to several root causes and not a single root cause. However, the differences among the projects signalised different factors in the organisation might influence why some defects require extensive effort to correct. This study was not designed to uncover which of these factors influence the software defect correction. In order to determine which factors influence what parts of the software defect correction process, more research based on the classical experiment design is needed.

Analysis of Defects

"It is hard to determine the location of the defect" were directly related to process of analysing the defect. This implies there were an inconsistency between the symptom and the root cause of the defect as stated by Eisenstadt in his survey of software defect war stories from developers in [Eisenstadt, 1997]. The comment discussions in the defect reports of this category were characterized with discussion of stack traces produced by crashes. Another main problem was lack of steps on how to reproduce the defect in the defect reports. Many defect reports did not contain these steps, and thus made it harder for developers to correct the defects as they were unable to reproduce the defects. This was in line with the study of Bettenburg et al. where they discovered which information was considered helpful in defect reports for software developers [Bettenburg et al., 2008]. The steps to reproduce the defect were considered among one of the most important helpful parts of a defect report for software developers. Much effort was used on making the defect reproducible through steps when these details were omitted from the original defect report.

Decisions

"Long clarification and discussion of the defect" were related to inter-developer communication. We were not able to find any studies suggesting this as a reason for defects requiring extensive effort to correct. However, in contrast to other organisations, developers are allowed to have discussions regarding the defect in the defect tracking system. We found no such functionality in the defect tracking system of another company in [Li et al., 2010a]. The comments in the defect reports in Company X gave us an unique view on the discussions, and it showed that prolonged discussions were contributing to increased effort to correct a defect. Research on communication in organisations have shown that employees prefer informal communication over formal communication when tasks are diverse and when they require group members to solve tasks [Ven et al., 1976]. Argote showed that groups engage in unscheduled meetings and communication when they are facing tasks with high uncertainty, and that they are more successful in these situations when they are relying on informal rather than formal communication [Argote, 1982]. We believe this is why this root cause has not been observed in the research performed because the communication during the process of correcting a defect is seldom recorded. Another aspect is the focus of our study, we focused on analysing the comments in defect reports. Other studies which have relied to defect tracking system data have used quantitative attributes of defect reports such as date and time stamps [Kim and Whitehead, 2006], number of tasks in the system [Lucia et al., 2005], attributes from classification scheme [Chillarege et al., 1992]

or effort data [Weiss et al., 2007; Hassouna and Tahvildari, 2010].

Ripple Effects

The original fix introduce new defects or multiple fixes. This phenomenon of introducing defects through correction of defects have been given attention in the research community as regression testing. The purpose of regression testing is to re-establish confidence that the modified code work as intended [Leung and White, 1989]. Regression testing is one of the most commonly used software testing techniques [Onoma et al., 1998]. Onama, Tsai, Poonawala, and Suganuma describes the costs associated with regression testing as the effort spent on developing new test cases, re-validating test cases, execution of the test suite, comparing test results and tracking test failures [Onoma et al., 1998]. Combined, these activities could impose significant costs on an organisation due to the total number of test cases could be large. On the other hand, it could be argued that introducing an incorrect defect correction is due to a inconsistency between symptom and root cause of the defect as Eisenstadt noted in his study [Eisenstadt, 1997]. Thus, the developers may perform an incorrect correction of the defect.

Jones state that incorrect corrections of defects were discovered in almost every company and project where incorrect corrections were studied [Jones, 2008b]. IBM discovered how 20 percent of all customer reported defects of one of their operating systems were incorrect corrections. On the other hand, approximately 5 percent of the defects in a application larger than 1000 function points will be incorrect fixes. This number increases with the complexity of the application and could reach 60 percent [Jones, 2008b].

Missing Functionality or Rewrite of Functionality

These defect reports were created due to missing functionality or the need to re-implement already existing functionality. Development of software is known to be costly. Jones uses the function point metric to describe software development costs in [Jones, 2008b]. The costs of implementing functionality which is equivalent to 100 function points is shown in Figure 6.1. A system considered to have 1000 function points and written in C could contain between 60000 and 170000 lines of code [Jones, 2008b]. From this, it is reasonable to argue that cost of implementing missing functionality or re-implementing functionality can be a significant high number. On average, implementing 100 function points of software costs the U.S. industry USD 79536.

We have not found this root cause in any of the research investigated prior to conducting this study. Possible reasons for this could be because of how the current

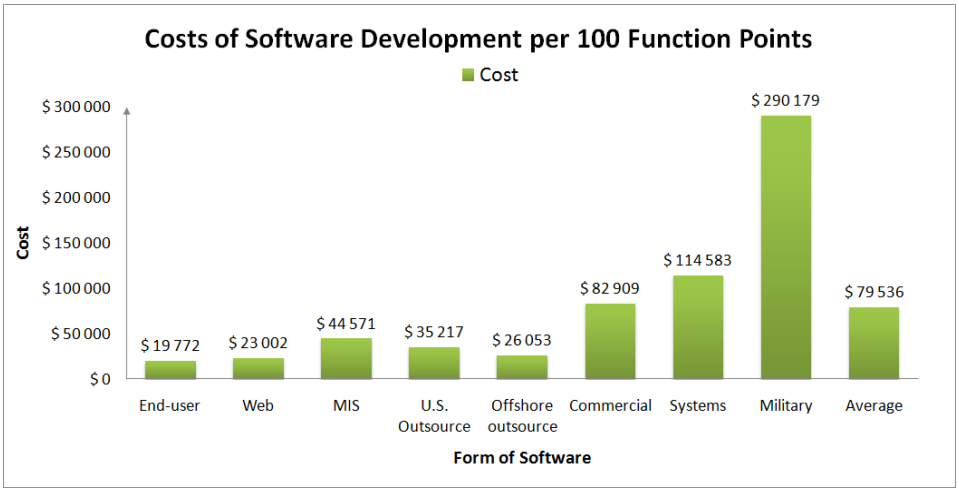


Figure 6.1: *The chart shows the costs of software development per 100 function points. The chart is visualised based on data from Jones [Jones, 2008b, page 314].*

research we investigated has been carried out. The research we investigate measured how certain factors influence the effort required to correct defects, they did not consider the root causes of why the defects required extensive effort to correct.

Differences Between Project Types

The results depicted a difference with regard to root causes for the core, B2C and B2B-projects. The frequencies of each category which was developed can be seen in Figure 5.2, and their portions in percent can be seen in Figure 6.2. The first main difference was that there were few B2B defect reports. The result of this can be seen in Figure 6.2 where small numbers have an impact on the portions size for each category in B2B-projects. However, we did not have access to many B2B-project defect reports due to legal concerns between Company X and their customers. B2B-projects defect reports were either categorised as *Hard to determine the location of the defect* or *Implemented functionality was new or needed a rewrite*, and have the largest portion of all project types in both categories, but the lowest frequencies in these categories for each of the project types.

However, the categorisation of the defect reports of the B2B projects among either *Hard to determine the location of the defect* or *Implemented functionality was new or needed a rewrite* seems to be natural according to the nature of the project. B2B projects concerns the implementation of the software of Company X on the customer platform, which would regularly be new hardware and new operating systems. Thus, defects classified as *Implemented functionality was new or needed*

a rewrite were expected as the platform the project were executed on could be new.

On the other hand, a new platform and operating system might influence the technical variables associated with development. Therefore, analysis and isolation of reported defects could become difficult. Research have shown that the experience of software developers with regard to both the software and the domain of the software influence the effort required to analyse, isolate and correct defects [Li et al., 2010a].

In contrast to B2B projects, 27,1 percent of the core project defects required extensive effort to correct due to discussions, while the B2C project had 5,2 percent within the same category. The B2C and B2B projects were concerned with implementation of the product line on different platforms to end users or customers, whereas the core project was concerned with implementation of the technology which forms the basis of B2C and B2B projects. The market they enter have fierce competition and there exists an incentive to be the first provider of a new feature or technology in the market. This incentive corresponds to a differentiation strategy in the Porter's framework for achieving competitive advantage in the market [Porter, 1998]. A strategy based on differentiation tries to achieve competitive advantage over competitors by introducing aspects which portrays the product as unique in the market place. On the other hand, according to D'aveni's model of hypercompetition, organisations continuously seek to establish new competitive advantages over competitors which will be eroded in the future [D'aveni and Gunther, 1994]. The framework and model of Porter and D'aveni give Company X and their competitors incentive to implement functionality from specifications which currently are working drafts. In addition, these specifications are specified by a non-biased third party organisation. As a result, much effort was spent on interpretation of requirements, discussion of how said requirements should be implemented, or discussion of technical solutions.

The higher rate of ripple effects compared to other projects was another aspect of the core-project. The total portion of defects that were ripple effects was 18,4 percent from the core project, and 9,4 percent from the B2C project. This difference was unexpected as code reviews were observed in the core project, but not in the B2C or B2B projects. Code reviews are a tool which can be used to reduce the frequency of incorrect corrections of defects [Jones, 2008b]. However, during the management meeting which took place on the 23th of March 2010, this observation was discussed. The representatives from Company X informed that the problem of ripple effects were known. Code reviews were introduced recently as an improvement in order to reduce ripple effects after an internal analysis by the project participants. However, the qualitative analysis supports the decision of the core project team to introduce code reviews in order to lower ripple effects. Nev-

ertheless, we could find no particular reason for why incorrect corrections should take place more frequently in the core project than the B2C or B2B projects. A reason for this difference could be the main development foci of the projects. Core project is not only concerned with incorporating new technology, but is also responsible for keeping the core functionality compatible with new functionality. This situation might induce more incorrect corrections than other projects, as they are mainly concerned with new code. Hence, our recommendation is to introduce code reviews in the B2C and B2B projects as these projects have ripple effects from defects too.

The main root causes of the B2C project were *Hard to determine the location of the defect* and *Implemented functionality was new or needed a rewrite*, accounting for approximately 70 percent of all the defect reports. However, the missing functionality or rework required were larger in the B2C project than the core project. This could be explained through the focus of the B2C project which are implementing new functionality in their flagship products for end users. The B2C project is where Company X faces the fiercest competition in the market. Time-to-market is an important factor for development of new products, and products need to be at the market at the right time, with the right price and the right quality [Minderhoud and Fraser, 2005]. This can be explained through D'aveni's model of hypercompetition [D'aveni and Gunther, 1994] where organisations seek to establish temporary competitive advantages. As a consequence of the requirement of a short time-to-market, software which is not sufficiently tested may hit the market. A survey performed in American software companies by Osterman Research found that 60 percent of defects were due to insufficient testing prior to software release [Inc., 2010]. These results were similar to the current situation at Company X where the problems can be related to insufficient testing because of time-pressure on developers. The software release time is influence by factors such as software reliability, costs of testing, and current market situation [Xie and Hong, 1998]. Time pressure may facilitate or inhibit creativity because of social and organisational factors Amabile et al. [2002]. In order to increase creativity, staff should feel like they are on a mission,that the work they perform is vital, and distractions and interruptions of the organisation should be minimised.

The B2C-project had 11.3 percent larger portion of *Reasons are not clear* than core-project. The defect reports were assigned in this category only when there were not possible to derive a root cause from the comments, description or title of the defect report.

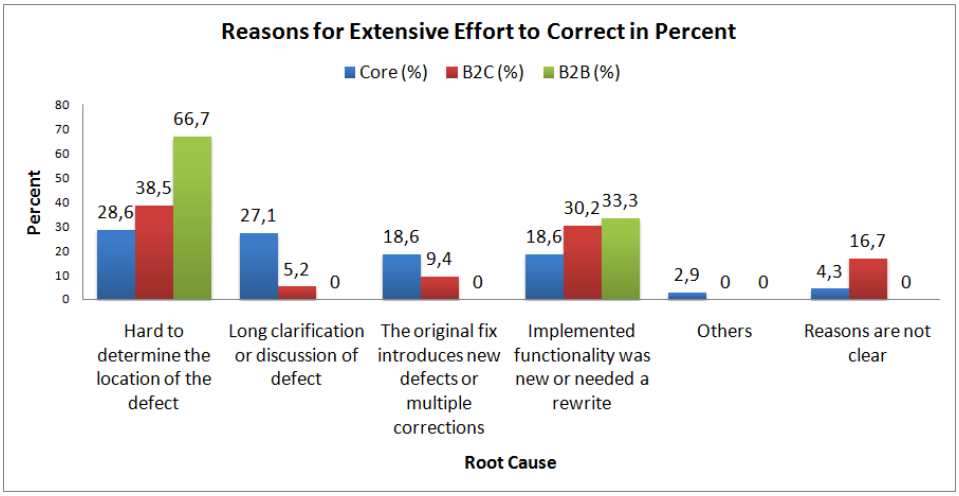


Figure 6.2: *The chart shows the result of the qualitative analysis in percentages instead of frequencies.*

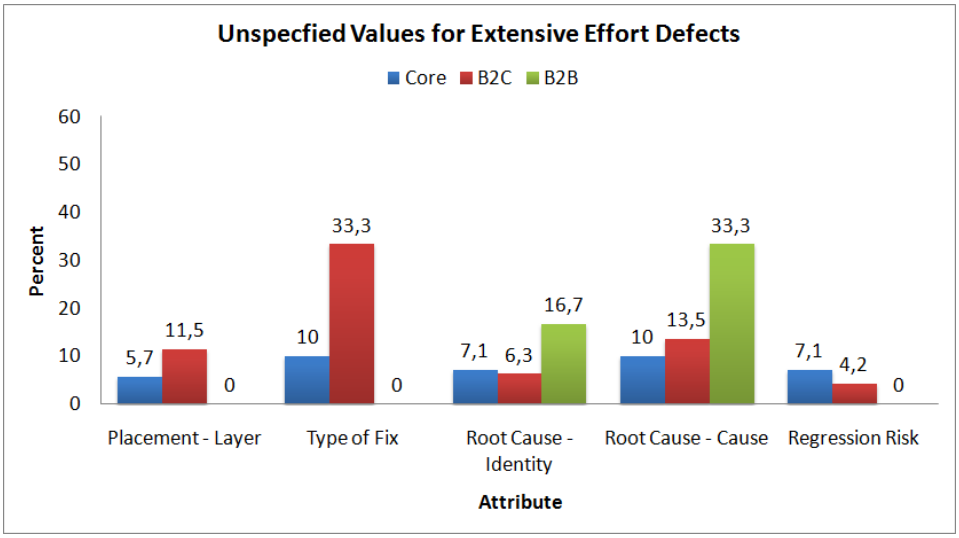


Figure 6.3: *The chart shows the portion of unspecified values in percent for each ODC-attribute from the defect reports which required extensive effort to correct of the three project types core, B2C, and B2B.*

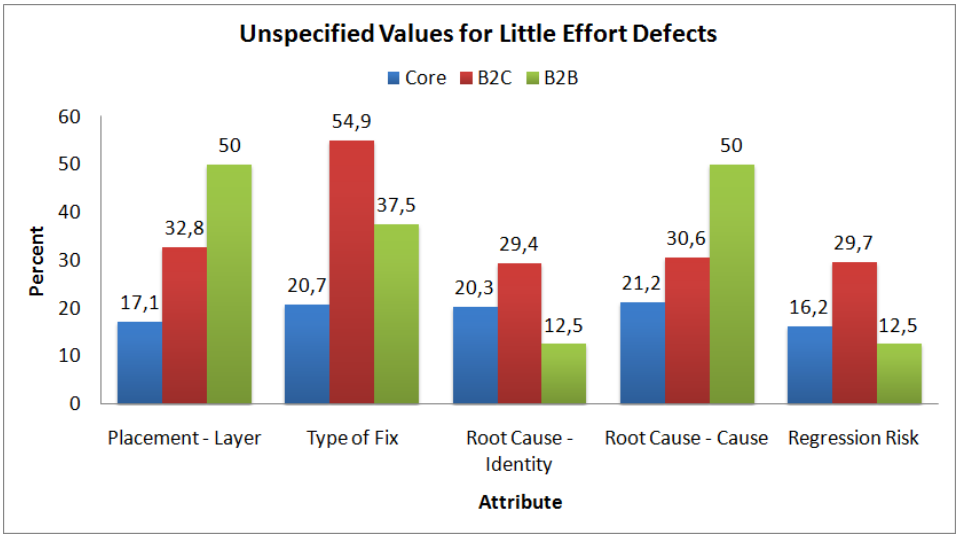


Figure 6.4: *The chart shows the portion of unspecified values in percent for each ODC-attribute from the defect reports which required little effort to correct of the three project types core, B2C, and B2B.*

Other Observations

Code Reviews were conducted in the core-project. No observations of code reviews were observed in the other projects. However, the study was not performed in such a way which could measure the effect of code reviews on projects, and further studies would be required in order to determine this. Code reviews is considered to be a good practice as frequent software inspections have been documented to be efficient at locating defects and incorrect corrections [Jones, 2008b].

A discussion in a defect report contained statements from an employee which considered the defect tracking system as a black hole of information. In other words, information enter the system in form of defect reports, but they are never used again in the future. Approximately 262600 defect reports were created in the defect tracking system since 1999, which is 71 defect reports every day in a 10-year period. Figure 6.5 shows how many defects which were reported every month in the B2C project throughout the period June 18th 2000 to June 15th 2010. Every defect report which is entered into the defect tracking system is a potential defect, and must thereafter be analysed. This process requires substantial effort due to the sheer amount of defects. The author found that 81.1% of the defects reported between November 2008 and November 2009 were reported by users. However, most of these defects were either duplicates or deemed to be invalid defects [Kris-

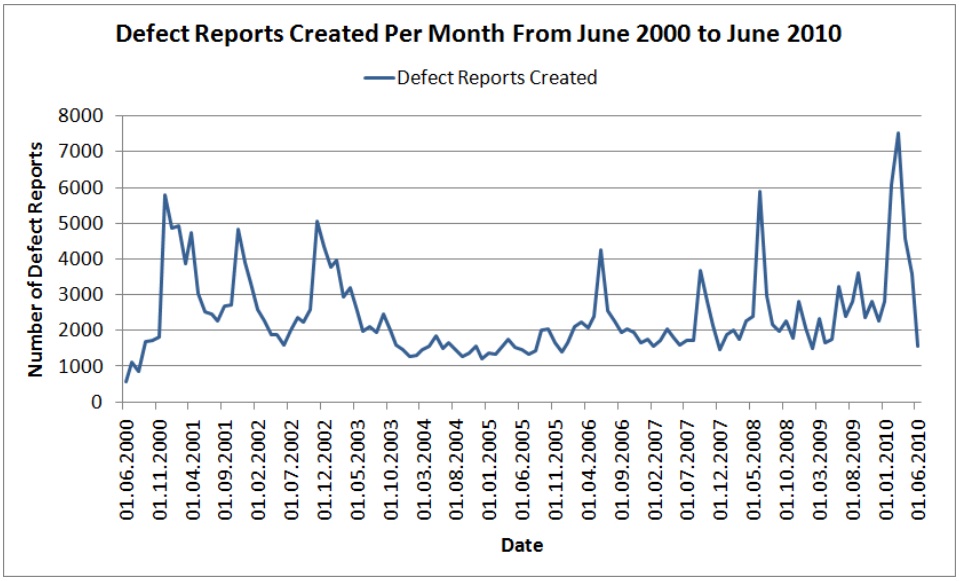


Figure 6.5: *The line chart shows number of defect reports created every month from June 18th 2000 to June 15th 2010.*

tiansen, 2009]. This is a process where substantial effort could be saved if the process were automated.

6.1.2 Results from Quantitative Analysis

The following sections will discuss the results obtained from the quantitative analysis.

Placement – Layer

The distribution of values for the attribute *Placement – Layer* could be described by the foci of the projects for the defects which required extensive effort to correct. Figure 3.1 in Chapter 3 gives an high-level description of how the values of the attributes were distributed between the three project types. Core project defect report values of *Placement – Layers* were related to the core-layer as depicted in Figure 5.3.

On the other hand, the *Placement – Layer* values of the B2C project defect reports were concerned with user interface and platforms. The B2C project is concerned with development of software which uses the core-project as a basis, and the software is developed for four main platforms. However, several issues were discov-

ered that were not related to issues in the B2C project, but needed to be propagated to the core project. The distribution of *Placement – Layer* values is consistent with Figure 3.1 where the core project is responsible for implementation of functionality which serves as a basis for products against different markets for B2C and B2B projects.

The B2B projects followed a similar distribution among the values of *Placement – Layer* as B2C project. However, a larger degree of the defects were propagated to the core project. The B2B projects are concerned with development and adaption of the core project on different devices. The multiple-device nature of B2B projects discover a larger degree of defects needed to be handled by the core project than the B2C project. The B2B projects were implemented on different platforms than the core and B2C projects, which explain why a higher portion of defects need to be propagated to the core project. The distribution of *Placement – Layer* values is consistent with Figure 3.1 and shows a similar pattern as the B2C project.

The defects which required little effort to correct followed a similar distribution as the extensive effort to correct defects. Nonetheless, the portion of defects which required to be propagated to the core project for correction were smaller. This behaviour can be explained through the reason why these defects require little effort. In [Kristiansen, 2009], we found that Defects which require little effort to correct were the result of errors in functions, classes, objects, or statements, and were injected later in the software development life cycle. In contrast, defects which required extensive effort to correct were result of algorithms, methods, memory management or concurrency management, and those defects which were injected earlier in the software development life cycle. Defects which require little effort to correct have in general less impact on the software system than equivalent defect which requires extensive effort to correct. Hence, the correction of defect which require little effort was more likely possible to perform locally, rather than propagating it further to a lower-level project like core than defects which require extensive effort to correct.

Type of Correction

The dominant type of correction in the core and B2C projects were corrections based on algorithms and methods. That was followed by checking statements for the core project, and functions, class, and objects for the B2C project. In contrast, the dominant correction in B2B projects were assign and initialisation statements. The differences of the core and B2C project were in line with the findings of Eisenstadt where hard to correct defects were related to algorithms and memory management [Eisenstadt, 1997]. The B2B projects were not consistent

with the findings of Eisenstadt.

However, a reason for this difference could related to the platform. B2B projects are developed on other platforms and devices than core and B2C project which are developed on windows, linux and MacOSX. The reason why assign and initialisation is a larger issue in B2B projects could be a cross-platform issue. As a result, different environmental variables which could lead to incorrect assignation or initialisation of variables in the code. Cusumano describes two different strategies for cross-platform development of software in [Cusumano and Yoffie, 1999]. The first strategy is to separately develop platform specific code from scratch for every different operating system which the software will run on. The second strategy is to write generic cross-platform code and tailor the generic code to the target operating system. Company X employs the second strategy by creating cross-platform software in the core project and tailoring it and development of platform specific user interfaces with the B2B and B2C projects. This could be the reason why there was a difference between core, B2C and B2B with regard to type of corrections.

There were differences between project types when we compared the corrections performed on defects which required extensive and little effort to correct. Figure 5.4 and Figure 5.10 showed these differences. The differences showed a trend where defects which required little effort to correct to be more code oriented compared to defects which required extensive effort to correct. The little effort to correct defects where related to program statements regarding conditional checking, function, classes, and objects. It is easier to analyse, isolate and correct defects which are directly visible in the code according to Eisenstadt [Eisenstadt, 1997]. In other words, there is not a inconsistency present between the root cause and the symptom of the defect. This inconsistency between the root cause and the symptom of the defect were present in hard to correct defects in the study of Eisenstadt [Eisenstadt, 1997].

Root Cause – Identity

There were significant differences between the project types with regard to *Root Cause – Identity*. The core and B2C projects followed a similar distribution. The only differences were that the core project had a larger portion of defects with origin in the design phase than B2C, and the B2C had a larger portion of defects with origin from the code phase than the core project. However, the B2B projects had a significant larger portion of defects with origin from the requirements phase. The desired functionality in the core and B2C projects are specified in the organisation. However, many of the requirements specifications for general software technologies are specified by third parties. Company X embrace open standards in their

software and these specifications are a central part of their implementation of new technologies.

On the other hand, Christel and Kang classifies the problems of requirements elicitation into three categories [Christel and Kang, 1992]:

- **Scope** is concerned with the scope of the requirements. Factors which should be considered when eliciting requirements are organisational, environmental, and project factors. The scope of the requirement should not be too board or too narrow.

- **Understanding** is concerned with taking all stakeholders information and arguments into account when eliciting requirements to avoid misunderstandings of requirements.

- **Volatility** is concerned with handling the change of requirement because of organisational, environmental or project factors.

The three categories of problems with requirements elicitation are all relevant for Company X. However, in the qualitative analysis, there were occasions were specification writers were asked questions to help interpretation of requirements. In addition, volatility is a risk as B2B-projects are under some circumstances developed on prototype devices.

The most significant difference between the identities of root causes for defects which require little or extensive effort to correct were requirement for B2B projects. The majority of the defects which were easy to correct were injected in the requirements phase as seen in Figure 5.11. This comparison contradicts the assumption of defects which require little effort to correct is injected during the coding phase and concerns program statements such as conditional checking, functions, classes and objects which was found in [Kristiansen, 2009]. There is no rule which governs how expensive a defect is to correct based on in which phase of the software development cycle it was injected. However, according to literature [Boehm and Basili, 2001], a defect is more likely to be expensive to correct the earlier it is injected to the process. Even though, a defect which is injected in the requirement phase could be a minor correction, for instance, from a typographical error in the requirements specification. On the other hand, there were 8 defect reports from B2B projects which were considered to be easy to correct. The individual defect report can influence the data to a higher degree. Other issues could be incorrect classification by the developer responsible for correcting the defect. However, this issue were a risk in any of the classified defect reports.

Root Cause – Cause

The analysis of the attribute *Root Cause – Cause* showed there were differences between the core project, and the B2B and B2C projects. The differences showed that the majority of core project defects were due to incorrect implementations, while the B2C and B2B projects had a larger portion of defects which were due to missing implementation. This situation can be related the project structure presented in Figure 3.1. The B2C and B2B projects are more concerned with development of new functionality in the products, while the core project is concerned with maintenance of existing to a larger degree and development of new functionality to a lesser degree.

Regression Risk

The core project had the most high regression risk corrections of defects, while B2B projects had the most medium regression risk correction of defects. In comparison, the B2C project had the most low regression risk corrections. The attribute *Type of Fix* helps to explain why this difference existed. Current research have focused on the risk of regressions problem from two approaches. The first approach is investigation of the properties of the change, and the second approach is to investigate the properties of the changed code [Mockus and Weiss, 2000]. Mockus and Weiss found that the risks of regressions were predicted by the properties of the change such as number of subsystems affected, the duration of the correction process, the experience of the software developer, and the number of lines of code added in the change [Mockus and Weiss, 2000]. Tarvo found that software metrics such as change metrics, code metrics, dependency metrics, and experience of software developer metrics [Tarvo, 2008]. On the other hand, Nagappan, Ball and Zeller found than the risks of regression correlated with the complexity metrics of the code, but no complexity metric proved to be better at predicting regressions than others [Nagappan et al., 2006]. First, we could argue the higher risk of regressions in the core project were due to changes in core modules could induce changes in other projects which used the modules. Second, experience could explain why regressions risks were higher in B2B projects than B2C projects. B2B projects use platforms which a software developer could be inexperienced with, while the B2C project develop software at a specified set of platforms. Last, the B2C project and B2B projects are more concerned with more implementation of new functionality than the core project.

There were a significant difference between defects which required extensive effort or little effort to correct. The risks of defects which required little effort to correct were lower than corresponding extensive effort to correct defects. This difference

can be explained through the scope of defects which were corrected. It is reasonable to assume that defects which require little effort to correct have lower values on code complexity, dependency and experience metrics. Hence, the defects have less risk of regressions according to [Mockus and Weiss, 2000].

Severity

The severity attribute had the same issues as discovered in [Kristiansen, 2009]. All of the B2B projects defect reports had the default value, while over 54 percent of the core and B2C attributes had the same value. The majority of the specified values of the *Severity* attribute were regarding specification violations, software crashes, and site compatibility issues for the core project. For the B2C project, the majority of the specified values were regarding crashes or other severe. The same categories for each attribute were dominant for the defects which required little effort to correct.

There are multiple problems with this attribute. First, the attribute is unspecified in over 50 percent of the cases because it have a default value as *Significant*. The author found that default values can influence the results obtained from attributes. Hence, default values in attributes should be avoided [Kristiansen, 2009]. Second, the categorises which is the reporter is able to select from the attribute have different abstraction levels. The values attribute can have is listed in Table 3.3. The contrast can be shown by comparing the categories *Trivial* and *Significant* with *Crashes the software* and *Blocks testing*. Both of the latter could be argued to be either significant or trivial. Last, the information derived from the attribute is indirectly provided elsewhere. Severity can be looked upon as a tool for prioritisation of which defects should be corrected. However, the defect tracking system already have two attributes concerning prioritisation of defects. The *Priority* attribute is given as a direct measure of how important the current issue is, while the *Customer Priority* attribute specifies the priority of the defect by a third party customer.

Unspecified Values

Figure 6.3 shows that the B2C-project had a percentage difference above 3 percent in 3 of 5 attributes for defects which required extensive effort to correct. Similarly, the same trend can be seen in Figure 6.4 where B2C attributes have at least a 9 percent larger portion of every attribute than core-project attributes. From these arguments, it is reasonable to assume the core-project developers have a more structured approach against defect analysis and classification schemes than B2C-project developers.

In addition, a larger portion of every attribute were unspecified in defects which have been classified as requiring little effort to correct than extensive effort. This assumption was determined to be true held when we compared the charts in Figure 6.3 and Figure 6.4. In the context of Company X, they have trouble with defects which require extensive effort to correct, and this problem were the main foci of the defect tracking system improvement which was introduced[1]. For this reason, we could argue that personnel responsible for updating defect reports pay more attention to specify accurate details in defect reports of defects which require extensive effort to correct. This phenomenon could be due to the greater potential of rewards if the same defect type could be avoided in future software development. However, this is speculative, and more data and other research strategies is required in order to verify this claim.

Another issue were default values in defect attributes. We found that the values of attributes were significantly influenced by default values if they were available in [Li et al., 2010b]. A default value is a value which is selected if no choice is made by the defect reporter. The *Severity* attribute had this problem in our data set. The result of this were that a single category dominated all other possible choices. We read through defect reports and found they should probably have been classified with another severity. As with in [Li et al., 2010b], we suspect defect reporters skip this value as it already has one assigned to it when they are creating the defect report.

6.1.3 Comparison versus Another Organisation

The results from the qualitative and quantitative analysis were compared against another company in [Li et al., 2010a]. Company Y is a large Norwegian software developer for the banking and financial sector. Currently, they have a staff of approximately 700 employees related to software development and quality assurance. We discovered the root causes of the defects requiring extensive effort to correct were different in Company X and Company Y. In Company Y, it was discovered that approximately 95 percent of the defects were due to lack of domain knowledge from developers. First, this further suggest results from the analysis is not generalisable to a general population, and is valid only for this specific organisation. This was why we picked the case study research design in order to be able to describe the current circumstances with software defects in Company X. Further, we wanted the possibility to compare the results with another organisation.

The results from the qualitative analysis showed other differences. In Company Y, we discovered a significant share of the defects which required little effort to

[1] See slide 13 of the presentation in Appendix C.

correct were due to defects in the graphical user interface, misinterpreted requirements or defects in the test environment the software developer is running [Li et al., 2010a]. In contrast, defects which required little effort to correct in Company X were different in each of the three projects types core, B2C and B2B. In the core project, the easy to correct defects were concerned with assignation, initialisation of variables, checking statements, had a lower regression risk and were injected during the code phase. In the B2C project, easy to correct defects were more concerned with checking statements in the code which had a low regression risk. Last, in the B2B projects, defects which required little effort to correct were due to checking statements, interfaces with third party libraries, had a lower regression risk and stem from requirements.

Similarly, the differences were found when we compared the defects which required extensive effort to correct from Company X and Company Y. In company Y, a third of the extensive effort defects were classified as logical errors in the code, missing functionality, and were injected during the design phase. In company X, there were differences between the project types. In the core project, extensive to correct defects were due to incorrect algorithms or methods, they were injected during the design phase, and had a high risk of regressions. In the B2C-project, the extensive effort to correct defects were due to algorithms, methods, functions, classes and objects. The defects were concerned with the core, platform, and user interface layers. The defects were injected during the design phase, and were due to either incorrect implementation or missing functionality. The defects from the B2C-project had lower regression risks than the core-project, but still higher than the little effort to correct defects. In the last project, the B2B-project, the defects which required extensive effort to correct were due to assignation and initialisation of variables, or function, classes and objects. A large portion of the B2B-project defects were related to the core-layer, and were injected during the coding phase. The defects had a average regression risk of medium.

These differences described above illustrate there are differences among organisations with regard to why defects require little or extensive effort to correct. In addition, it demonstrates there are differences within an organisation based on different project types. The effort required to correct defects are based on both technical, social and organisational factors [Aranda and Venolia, 2009]. Based on the comparison above, it is reasonable to argue that an organisation should adapt processes based on their own needs, and regularly monitor the progress of projects with regard to software quality. Literature suggests introducing improvements through pilot-studies before widespread organisational adaption of the improvement take place [Stålhane, 2008]. The improvement should be planned, implemented, checked for results and acted upon, corresponding to a Shewhart-cycle as

described in [Shewhart and Deming, 1986]. However, it is important to note that improvements which are selected for widespread adoption in the organisation are monitored after implementation due to the internal differences illustrated above. In addition, when searching for new improvement opportunities, the old improvement must be adjusted in the analysis, due to these having a dynamic impact on the organisation [Li et al., 2010a].

6.1.4 Current Research Versus This Study

Current research performed in this area did not try to derive root causes for why a certain amount of software defects take extensive effort. However, they modelled the effort required to correct the software defects by choosing an arbitrary set of software metrics. These software metrics were chosen since they were thought of having an influence on the effort required to correct the defect. The predicted effort from the developed model was compared to the actual effort, and how well each software metric predicted effort required were measured.

The main difference between this study and existing studies is how we developed the root causes. We developed the root causes through use of grounded theory. The comments and discussions in the defect reports were coded, further developed to concepts, and then to categories. The categories presented as results here are derived from the discussions in the defect reports. The study from Eisenstadt are the closest study to our study in terms how he derived his results in [Eisenstadt, 1997]. However, Eisenstadt analysed defect correction stories which were written down and collected post-correction of the defects. The discussions consisting of comments analysed in our study were written and collected during the defect correction process.

6.2 Validity Threats

This sections discusses the validity of this study according to the aspects specified in Section 4.4. The evaluation of validity against the criteria suggested by [Wohlin et al., 2000]:

- **Internal Validity:** The source of the data is from the defect database of company X and we see two threats to internal validity. The first threat to internal validity is that defect reports might be incorrectly classified by defect reporters and developers. The defects have been read through and the few places where we found obvious mistakes have been corrected. The second threat is misunderstanding of discussions in the comment attribute in the defect reports. As the candidate is an external observer, he might not

have sufficient understanding of internal processes. However, this issue was addressed by presenting and discussing the results with representatives from Company X during a meeting in mid-March.

- **Construct Validity:** The quantitative analysis used frequencies in tables, and difference of portions were used to determine differences. Hence, there was no use of statistical tests and no statistical significant relationships were established. The study was performed in a descriptive manner. The qualitative analysis followed guidelines set by [Shannak and Aldhmour, 2009]. However, a central concept of grounded theory is to generate a theory from the data [Strauss and Corbin, 1998]. This is in contrast to quantitative analysis which sets test a theory against the data.

- **External Validity:** This study was descriptive of Company X. The comparison between the studies performed at Company X and Company Y in Section 6.1.3 illustrates how the costs of correction extensive defects are different in each organisation. Hence, one should be careful with generalising the results to another organisations.

- **Conclusion Validity:** We see no threat to conclusion validity of the qualitative analysis due to it followed the guidelines set by [Shannak and Aldhmour, 2009]. The results obtained were discussed and compared to findings of other researchers in both the qualitative and quantitative analysis. All of our assumptions are stated and justified. Hence, we see no threat to the conclusion validity of this study.

The validity of the qualitative study was evaluated against the criteria suggested by [Trochim and Donnelly, 2006]:

- **Credibility:** the results of the qualitative analysis were presented to the EVISOFT-responsible at Company X during a meeting on March 23rd 2010. The present personnel from Company X found the results to be believable and told it reflected the current situation after discussion.

- **Transferability:** The results from the qualitative analysis is valid for Company X. The results should not be generalised to other contexts without careful considerations.

- **Dependability:** The data were collected during one evening where little changes were done to the defect tracking system. The collected data were defect reports and the information gathered pose no threats from subjective bias from the author.

- **Confirmability:** The data set have been analysed by the author. However, Li analysed parts of the data set during the autumn of 2009 and reached similar conclusions. Hence, we see no threats to the confirmability of the study.

CHAPTER 7

CONCLUSION

The following chapter concludes the work performed during the writing of this thesis, and gives suggestions to further work based on the work in this thesis.

7.1 Main Contributions

This thesis has the following main contributions:

- Four main root causes for defects which require extensive effort to correct. These were *it is hard to determine the location of the defect*, *long clarification and discussion of the defect*, *the original fix introduces new defects or multiple fixes*, and *the implemented functionality was new or needed a rewrite*.
- Project profiles with regard to ODC attributes and differences between defects which require extensive or little effort to correct.
- Unspecified values are a concern in defect reports since attributes with no values provide incomplete information regarding the defect. And unspecified values are specified to a lesser degree for defects which require little effort to correct in contrast to defects which require extensive effort to correct.
- Default values influence the values of the attribute as discovered by [Kristiansen, 2009; Li et al., 2010b].
- Different organisations will have different distributions of root causes on different projects. However, within the same organisations will have different distributions of root causes for different project. A structured and systematic analysis of these differences yields opportunities for software process improvement.

7.2 Further Work

The following section suggests further work with regard to both specific topics of Company X and their EVISOFT-participation, and software defect research.

7.2.1 Topics Specific to Company X

Further work for the EVISOFT-project were planned during a meeting with representatives on April 29th. The minutes from this meeting can be found in Appendix D. The planned work related to this thesis is to present the results from this thesis to developers, quality assurance personnel, and project managers working in Company X. Other aspects related to the work with this thesis are improvement of the defect classification scheme, and further analysis of defect reports.

There are several other future activities planned for Company X. Such activities include duplicate defect detection, development of cost-benefit model for defect analysis, and improvement of defect management in the defect tracking system.

7.2.2 Software Defect Research

Several aspects are interesting to pursue. First, it could be interesting to do a cost and benefit analysis of the defect classification improvement. Literature states of defect tracking system information is one of the most valuable sources of information for software process improvement activities [Grady, 1996]. A cost and benefit analysis could pin-point interesting effects of the improvement, and possibly be used to gain further support for improvements. For instance, by increasing software developers motivation to Hence, facilitating minimisation of the unspecified values in defect reports.

More similar studies in different organisations in order to create a framework of factors for defect classification of root causes. This could possibly help organisations eliciting areas of improvement by helping them overcome similar challenges faced by other organisations. In addition, it would be interesting to study how and why identified factors influence the effort required to correct a defect, in order to establish this framework. The factors derived from defect reports could be used for defect correction effort estimation.

In general, more similar studies of this one would be interesting as the root causes of defects were determined to be different for different organisations. This could lead to establishment of common factors which influence the effort required to correct defects. As a result, a better understanding of how organisational, social and technical factors influence the effort required to correct defects.

7.3 Recommendations

This section provides advice to industry and academia. For industries, organisations seeking to do software process improvement should not use techniques as silver bullets. An organisation wishing to do software process improvement activities should analyse the current situation and deal with it accordingly. Graham et al. suggested that the software testing process is planned and adapted to the organisation's own needs [Graham et al., 2008]. We believe this is important for any software engineering activity. So specifically for using defect data for software process improvement activities, the following steps were suggested in [Li et al., 2010a]:

1. Classify randomly selected defects using a orthogonal defect classification scheme.
2. Analyse the defect rates.
3. Analyse the cost drivers and correction productivity for each category of defects.
4. Establish a profile of the project which can be compared to similar projects in the organisation.

Lieberman stated the lack of research of software defects were a scandal [Lieberman, 1997]. The survey of the the state-of-the-art showed there were lack of research on software defects, and the results obtained from the studies investigated were side effects of the study in question. Due to the economic considerations of software quality and defects in software [Jones, 2008b; Tassey, 2002], more research should be devoted to research of the nature of software defects. It is important to study the nature of defects, in order to create better processes for preventing and handling defects. Bertolini states the main challenge and goal of software testing research is development a unified software testing theory [Bertolino, 2007]. We see no reason for academia to not have the same goal for software defects.

7.4 Conclusion

We have given a brief overview of the fields of software engineering, software maintenance, software defects and software verification. The literature study showed there were a lack of research of the software defect phenomenon directly, but there were diversity in the research of fields which software defects affected.

The research design selected were a descriptive case study of Company X. We used qualitative methods on the discussions in defect reports in order to establish root

causes for why defects required extensive effort to correct. In addition, we used quantitative methods on the orthogonal defect classification attributes in the defect reports to compare defects which required extensive or little effort to correct. The orthogonal defect classification attributes were extracted using a script developed during master fall project of the candidate.

The qualitative analysis derived four main root causes which explain why a group of defects require extensive effort to correct. The first root cause were that it were hard to determine the location of the defect. The second root cause were related to discussion among developers among concepts which required clarification in order to correct the defect. The third root cause were due to incorrect corrections of defects which introduced new defects into the system. The last root cause were implementation tasks of new or existing functionality which needed a rewrite. In addition, two defect reports were the reason of communication or documentation issues, while we were unable to determine the root cause of several defect reports due to limited discussion. One of the root causes were determined to be new contributions and not described in existing literature surveyed in the literature study. It was the root cause which were concerned with implementation tasks of new or existing functionality which needed a rewrite.

A comparison between the four root causes and project types revealed the root causes were influenced by the project types. The core-project had a larger degree of discussion and incorrect corrections than B2C and B2B-projects. In contrast, the B2B and B2C-projects were concerned with hard to locate defects and implementation of missing functionality or re-implementation of functionality. Similarly, a comparison against another organisation showed there were differences with regard to root causes for extensive effort. In company Y, the main root cause were insufficient domain knowledge. This showed how systematic analysis of defect reports can yield software process improvement opportunities.

The quantitative analysis uncovered differences among extensive or little effort to correct defects, and differences among project types. In the core project, extensive effort to correct defects were due to incorrect algorithms or methods, they were injected during the design phase, and had a high risk of regressions. In the B2C-project, the extensive effort to correct defects were due to algorithms, methods, functions, classes and objects. The defects were concerned with the core, platform, and user interface layers. The defects were injected during the design phase, and were due to either incorrect implementation or missing functionality. The defects from the B2C-project had lower regression risks than the core-project, but still higher than the little effort to correct defects. In the last project, the B2B-project, the defects which required extensive effort to correct were due to assignation and initialisation of variables, or function, classes and objects. A large portion of the

B2B-project defects were related to the core-layer, and were injected during the coding phase. The defects had a average regression risk of medium.

The little effort to correct defects in the core project were concerned with assignation or initialisation of variables, checking statements, and had a lower regression risk and were injected during the code phase. In the B2C-project, easy to fix defects were more concerned with checking statements in the code which had a low regression risk. Last, in the B2B project, defects which required little effort to correct were due to checking statements, interfaces with third party libraries, had a lower regression risk and stem from requirements. The data set for the quantitative analysis contained high levels of unspecified values for little effort to correct defect. The levels of unspecified attributes were lower for defects which required extensive effort to correct. A comparison of the quantitative data versus another organisation showed similarities with regard to defects which required little effort to correct. However, there were differences between the organisations with regard to extensive effort defects. In Company Y, the main cause were logical errors in code for defects which required extensive effort to correct.

In conclusion, we have shown how qualitative methods can be used to derive root causes. In addition, we have shown the main root causes of defects can be different within the same organisation as well as in other organisations, and how the extensive effort to correct defects were different from the little effort to correct defects.

GLOSSARY

agility "means to strip away as much of the heaviness, commonly associated with traditional software-development methodologies, as possible to promote quick response to changing environments, changes in user requirements, accelerated project deadlines, and the like." [Erickson et al., 2005, page 89]

bad fix see *incorrect correction*.

bug see *defect*.

code review A process where the code is read through by a qualified person which has been involved in the code in order to locate defects. Code reviews have been proven to be efficient to reduce the frequency of incorrect corrections of defects [Jones, 2008a].

debugging see *defect correction*.

defect (fault) "An incorrect step, process, or data definition in a computer program." [Society, 1990, page 32]

defect correction "To detect, locate, and correct faults in a computer program." [Society, 1990, page 25]

defect removal see *defect correction*.

dynamic analysis To verify software components during execution [Harrold, 2000, page 1]. Dynamic analysis techniques include functional, control flow, data flow, mutation and regression testing.

error "difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition." [Society, 1990, page 31]

failure "The result of a fault." (From [Society, 1990, page 32])

fault see *defect*.

fix see *correction*.

function points A measurement of the functionality provided to a user in software [Symons, 1988].

incorrect defect correction A correction of a defect which causes a regression [Jones, 2008a]

latent defect A defect which is not discovered before the release of the software [Jones, 2008a]

mistake A human action which produces the software defect. [Society, 1990, page 32]

quality "(1) The degree to which a system, component, or process meets specified requirements. (2) The degree to which a system, component, or process meets customer or user needs or expectations." [Society, 1990, page 60]

regression testing The process of re-evaluating modified code in order to locate where the defect occurred after the modification took place [Leung and White, 1989].

software engineering "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software." [Society, 1990, page 67]

software maintenance "the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment." [Society, 1990, page 46]

software reuse The process of developing software systems by use of existing software rather than starting from scratch each time [Krueger, 1992].

static analysis To verify source code not under execution through use of formal and informal methods [Harrold, 2000, page 1]. Static analysis techniques include abstract interpretation, model checking, deductive methods, code reviews and walkthroughs.

BIBLIOGRAPHY

Evisoft homepage. Website. `http://www.sintef.no/Projectweb/Evisoft/`, retrieved 08.12.2009.

David J. Agans. *Debugging*. Amacom, New York, NY, USA, 2006. ISBN 9780814474570.

Yunsik Ahn, Jungseok Suh, Seungryeol Kim, and Hyunsoo Kim. The software maintenance project effort estimation model based on function points. *Journal of Software Maintenance*, 15(2):71–85, 2003. ISSN 1040-550X. doi: http://dx.doi.org/10.1002/smr.269.

Teresa M. Amabile, Constance N. Hadley, and Steven J. Kramer. Creativity under the gun. *Harvard Business Review*, 80(8):52–61, 2002. ISSN 0017-8012. doi: http://dx.doi.org/10.1225/R0208C.

Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, United Kingdom, 2008. ISBN 9780521880381.

Jorge Aranda and Gina Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 298–308, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: http://dx.doi.org/10.1109/ICSE.2009.5070530.

Linda Argote. Input uncertainty and organizational coordination in hospital emergency units. *Administrative Science Quarterly*, 27(3):420–434, 1982. ISSN 0001-8392. doi: http://dx.doi.org/10.2307/2392320.

Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, Boston, MA, USA, 2003. ISBN 0321146530.

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber,

and Jeff Sutherland. Manifesto for agile software development. Website. `http://agilemanifesto.org/`, retrieved 09.15.2009.

Laszlo A. Belady and Meir M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976. ISSN 0018-8670. doi: http://dx.doi.org/10.1147/sj.153.0225.

Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: http://dx.doi.org/10.1109/FOSE.2007.25.

Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-995-1. doi: http://doi.acm.org/10.1145/1453101.1453146.

Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/2.962984.

Eric Braude. *Software Engineering: An Object-Oriented Perspective*. Jon Wiley and Sons, London, United Kingdom, 2001. ISBN 0471322083.

Gerardo Canfora and Aniello Cimitile. Software maintenance. *Handbook of Software Engineering and Knowledge Engineering*, 1:91–120, 2001.

Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.177364.

Michael G. Christel and Kyo C. Kang. Issues in requirements elicitation. Technical Report ESC-TR-92-012 CMU/SEI-92-TR, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

Michael A. Cusumano and David B. Yoffie. What netscape learned from cross-platform software development. *Communications of the ACM*, 42(10):72–78, 1999. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/317665.317678.

Richard A. D'aveni and Robert Gunther. *Hypercompetition: Managing the Dynamics of Strategic Manoeuvring*. The Free Press, New York, NY, USA, 1st edition, 1994. ISBN 978-0029069387.

QA Department. Bug states and workflow. Internal document.

Matthew B. Dwyer, John Hatcliff, Robby Robby, Corina S. Pasareanu, and Willem Visser. Formal software analysis emerging trends in software model checking. In *FOSE '07: 2007 Future of Software Engineering*, pages 120–136, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: http://dx.doi.org/10.1109/FOSE.2007.6.

Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9-10):833–859, 2008. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2008.01.006.

Marc Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40 (4):30–37, 1997. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/248448.248456.

John Erickson, Kalle Lyytinen, and Keng Siau. Agile modeling, agile software development, and extreme programming: The state of research. *Journal of Database Management*, 16(4):88–100, 2005. ISSN 1063-8016.

William M. Evanco. Modeling the effort to correct faults. In *Selected papers of the sixth annual Oregon workshop on Software metrics*, pages 75–84, New York, NY, USA, 1995. Elsevier Science Inc. doi: http://dx.doi.org/10.1016/0164-1212(94)00129-B.

William M. Evanco. Prediction models for software fault correction effort. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 114–120, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1028-0. doi: http://dx.doi.org/10.1109/.2001.914975.

Micheal E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986. ISSN 0098-5589.

Michael Fredericks and Victor Basili. Using defect tracking and analysis to improve software quality. Report DACS-SOAR-98-2, Experimental Software Engineering Group, University of Maryland, College Park, MD, USA, 1998.

Bernd Freimut. Developing and using defect classification schemes. Forschungsbericht IESE-Report No. 072.01/E, Fraunhofer Institut Experimentelles Software Engineering, Kaiserslautern, Germany, 2001.

David A. Garvin. What does “product quality” really mean? *MIT Sloan Management Review*, 26(1):25–43, 1984. ISSN 1532-9194.

Pär J. Ågerfalk and Brian Fitzgerald. Flexible and distributed software processes: old petunias in new bowls? *Communications of the ACM*, 49(10):27–34, October 2006. ISSN 0001-0782.

Robert B. Grady. Software failure analysis for high-return process improvement decisions. *Hewlett-Packard Journal*, 47(4):15–24, 1996. ISSN 0018-1153.

Dorothy Graham, Erik Van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing*. Cengage Learning EMEA, London, United Kingdom, 2008. ISBN 9781844809899.

Mary Jean Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0. doi: http://doi.acm.org/10.1145/336512.336532.

Alaa Hassouna and Ladan Tahvildari. An effort prediction framework for software defect correction. *Information and Software Technology*, 52(2):197–209, 2010. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2009.10.003.

Osterman Research Inc. Survey report for electric cloud: An osterman research survey report. May 2010.

ISO/IEC. *ISO/IEC 9126-1:2001 - Software Engineering – Product Quality*. Number 9126-1:2001. ISO/IEC, 2001. ISBN 0-580-36526-3.

Capers Jones. Software quality in 2008: Survey of the state of the art. Presentation, 2008a. `http://www.scribd.com/doc/7758538/Capers-Jones-Software-Quality-in-2008`. Accessed 01.05.2010.

Capers Jones. *Applied Software Measurement: Global Analysis of Productivity and Quality*. McGraw-Hill, New York, NY, USA, 3rd edition, 2008b. ISBN 978-0-07-150244-3.

Capers Jones. *Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies*. McGraw-Hill, New York, NY, USA, 2009. ISBN 007162161X.

Joseph M. Juran. *Juran's Quality Handbook*. Twayne Publishers, Boston, MA, USA, 1999. ISBN 9780070340039.

Natalia Juristo, Ana M. Moreno, and Sira Vegas. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, 9(1-2):7–44, 2004. ISSN 1382-3256. doi: http://dx.doi.org/10.1023/B:EMSE.0000013513.48963.1b.

Sunghun Kim and E. James Whitehead, Jr. How long did it take to fix bugs? In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 173–174, New York, NY, USA, 2006. ACM. ISBN 1-59593-397-2. doi: http://doi.acm.org/10.1145/1137983.1138027.

Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1):12–21, 1996. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/52.476281.

Jan M. W. Kristiansen. Using orthogonal defect classification in a norwegian software company. Master thesis project, December 2009.

Charles W. Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2): 131–183, 1992. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/130844.130856.

Craig Larman. *Agile and Iterative Development: A Manager's Guide*. Pearson Education, 1st edition, 2003. ISBN 0131111558.

Craig Larman and Victor R. Basili. Iterative and incremental development: A brief history. *Computer*, 36(6):47–56, 2003. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.2003.1204375.

Meir M. Lehman. Laws of software evolution revisited. In *EWSPT '96: Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, London, UK, 1996. Springer-Verlag. ISBN 3-540-61771-X. doi: http://dx.doi.org/10.1007/BFb0017737.

Hareton K. N. Leung and Lee White. Insights into regression testing. In *Proceedings of Conference on Software Maintenance*, pages 60–69, Los Alamitos, CA, USA, October 1989. IEEE Computer Society Press. ISBN 0-8186-1965-1. doi: http://dx.doi.org/10.1109/ICSM.1989.65194.

Jingyue Li and Qitao Gan. Proposal for defect analysis and bts improvement, 2009. Internal document.

Jingyue Li, Tor Stålhane, Jan M. W. Kristiansen, and Reidar Conradi. Cost drivers of software corrective maitenance: An empirical study in two companies. 2010a.

Jingyue Li, Tor Stålhane, Jan M. W. Kristiansen, and Reidar Conradi. Enhancing software defect tracking system to facilitate continuous software quality assessment and improvement. 2010b.

Paul Luo Li, Mingtian Ni, Song Xue, Joseph P. Mullally, Mario Garzia, and Mujtaba Khambatti. Reliability assessment of mass-market software: Insights from windows vista®. In *ISSRE '08: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 265–270, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3405-3. doi: http://dx.doi.org/10.1109/ISSRE.2008.60.

Henry Lieberman. The debugging scandal and what to do about it. *Communications of the ACM*, 40(4):26–29, 1997. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/248448.248455.

Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980. ISBN 0201042053.

Andrea De Lucia, Eugenio Pompella, and Silvio Stefanucci. Assessing effort estimation models for corrective maintenance through empirical studies. *Information and Software Technology*, 47(1):3–15, 2005. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2004.05.002.

Michael R. Lyu. *Handbook of Software Reliability Engineering*. McGraw Hill, Boston, MA, USA, 1996. ISBN 0070394008.

Michael R. Lyu. Software reliability engineering: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 153–170, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: http://dx.doi.org/10.1109/FOSE.2007.24.

Michael S. Mahoney. Finding a history for software engineering. *IEEE Annals of the History of Computing*, 26(1):8–19, 2004. ISSN 1058-6180. doi: http://dx.doi.org/10.1109/MAHC.2004.1278847.

George Marakas. *Systems Analysis and Design: an Active Approach*. McGraw-Hill, New York, NY, USA, 2006. ISBN 0072976071.

Simon Minderhoud and Peter Fraser. Shifting paradigms of product development in fast and dynamic markets. *Reliability Engineering and System Safety*, 88(2):127–135, 2005. ISSN 0951-8320. doi: http://dx.doi.org/10.1016/j.ress.2004.07.002.

Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000. ISSN 1089-7089. doi: http://dx.doi.org/10.1002/bltj.2229.

Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 452–461, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: http://doi.acm.org/10.1145/1134285.1134349.

Peter Naur and Brian Randell. Software engineering: Report of a conference sponsored by the nato science committee. Technical report, NATO, Brussels, Belgium, 1969.

Frank Niessink and H. van Vliet. Two case studies in measuring software maintenance effort. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 76, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8779-7. doi: http://dx.doi.org/10.1109/ICSM.1998.738495.

Briony J. Oates. *Researching Information Systems and Computing*. Sage Publications Ltd., London, United Kingdom, 2006. ISBN 1412902231.

Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41 (5):81–86, 1998. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/274946.274960.

Leon J. Osterweil. Strategic directions in software quality. *ACM Computing Surveys*, 28(4):738–750, 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/242223.242288.

Leon J. Osterweil. A future for software engineering? In *FOSE '07: 2007 Future of Software Engineering*, pages 1–11, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: http://dx.doi.org/10.1109/FOSE.2007.1.

David L. Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, 12(2):251–257, 1986. ISSN 0098-5589. doi: 10.1007/3-540-15199-0_6.

Hoang Pham. Software reliability and cost models: Perspectives, comparison, and practice. *European Journal of Operational Research*, 149(3):475–489, 2003. ISSN 0377-2217. doi: http://dx.doi.org/10.1016/S0377-2217(02)00498-8.

Adam A. Porter. Fundamental laws and assumptions of software maintenance. *Empirical Software Engineering*, 2(2):119–131, 1997. ISSN 1382-3256. doi: http://dx.doi.org/10.1023/A:1009793015685.

Michael E. Porter. *Competitive Strategy: Techniques for Analyzing Industries and Competitors*. The Free Press, New York, NY, USA, 1st edition, 1998. ISBN 978-0684841489.

Sam Ramanujan, Richard W. Scamell, and Jaymeen R. Shah. An experimental investigation of the impact of individual, program, and organizational characteristics on software maintenance effort. *Journal of Systems and Software*, 54(2):137–157, 2000. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/S0164-1212(00)00033-9.

Kristen Ringdal. *Enhet og Mangfold*. Fagbokforlaget, 2nd edition, 2009. ISBN 978-82-450-0569-1.

Dieter Rombach, Marcus Ciolkowski, Ross Jeffery, Oliver Laitenberger, Frank McGarry, and Forrest Shull. Impact of research on practice in the field of inspections, reviews and walkthroughs: learning from successful industrial uses. *ACM SIGSOFT Software Engineering Notes*, 33(6):26–35, 2008. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1449603.1449609.

W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 0-89791-216-0.

Bruce Schneier. *Secrets and Lies*. Wiley, New York, 2004. ISBN 0471453803.

Rifat O. Shannak and Fairouz M. Aldhmour. Grounded theory as a methodology for theory generation in information systems research. *European Journal of Economics, Finance and Administrative Sciences*, (15):32–50, 2009. ISSN 1450-2887.

Walter A. Shewhart and William E. Deming. *Statistical Method from the Viewpoint of Quality Control*. Dover Publications, Mineola, NY, USA, dover edition, 1986. ISBN 978-0486652320.

Ruchi Shukla and Arun Kumar Misra. Estimating software maintenance effort: a neural network approach. In *ISEC '08: Proceedings of the 1st India software engineering conference*, pages 107–112, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-917-3. doi: http://doi.acm.org/10.1145/1342211.1342232.

Sandra Slaughter and Rajiv D. Banker. A study of the effects of software development practices on software maintenance effort. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, pages 197–205, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7677-9. doi: http://dx.doi.org/10.1109/ICSM.1996.565007.

Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan. Evaluating the cost of software quality. *Communications of the ACM*, 41(8):67–73, 1998. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/280324.280335.

IEEE Computer Society. *IEEE Standard Glossary of Software Engineering Terminology: IEEE Standard 610.12-1990*. Number 610.12-1990 in IEEE Standard. 1990. ISBN 1-55937-067-X. doi: http://dx.doi.org/10.1109/IEEESTD.1990.101064.

IEEE Computer Society. *IEEE Standard for Software and System Test Documentation*. Number 829-2008 in IEEE Standard. 2008. ISBN 978-0-7381-5747-4. doi: http://dx.doi.org/10.1109/IEEESTD.2008.4578383.

Tor Stålhane. *Kompendie i TDT4235 Programvarekvalitet og prosessforbedring*. Tapir Akademisk Forlag, Trondheim, Norway, 2008.

Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, New York, NY, USA, 2nd edition, 1998. ISBN 0803959400.

Charles R. Symons. Function point analysis: Difficulties and improvements. *IEEE Transactions on Software Engineering*, 14(1):2–11, 1988. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.4618.

Alexander Tarvo. Using statistical models to predict software regressions. In *ISSRE '08: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 259–264, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3405-3. doi: http://dx.doi.org/10.1109/ISSRE.2008.21.

Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. Technical Report RTI Project Number 7007.011, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2002.

Robert Thibodeau. The state-of-the-art in software error data collection and analysis. Technical Report ADA075228, Battelle Columbus Labs, Columbus, OH, USA, 1978.

William Trochim and James P. Donnelly. *The Research Methods Knowledge Base*. Atomic Dog Publishing, Cincinnati, OH, USA, 3rd edition, 2006. ISBN 978-1592602919.

Andrew H. Van De Ven, Andre L. Delbecq, and Richard Koenig, Jr. Determinants of coordination modes within organizations. *American Sociological Review*, 41 (2):322–338, 1976. ISSN 0003-1224. doi: http://dx.doi.org/10.2307/2094477.

Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 1, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2950-X. doi: http://dx.doi.org/10.1109/MSR.2007.13.

James C. Westland. The cost behavior of software defects. *Decision Support Systems*, 37(2):229–238, 2004. ISSN 0167-9236. doi: http://dx.doi.org/10.1016/S0167-9236(03)00020-4.

Robin Whittemore, Susan K. Chase, and Carol L. Mandle. Validity in qualitative research. *Qualitative Health Research*, 11(4):522–537, 2001. ISSN 1049-7323. doi: http://dx.doi.org/10.1177/104973201129119299.

Claes Wohlin, Per Runeson, Martin Host, Magnus C. Ohlsson, Bjorn Regnell, and Anders Wesslen. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic, Boston, MA, USA, 2000. ISBN 0-7923-8682-5.

Min Xie and Guan-Yue Hong. A study of the sensitivity of software release time. *Journal of Systems and Software*, 44(2):163–168, 1998. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/S0164-1212(98)10053-5.

Robert Yin. *Case Study Research: Design and Methods*. McGraw-Hill, New York, NY, USA, 2003. ISBN 0761925538.

APPENDIX A

RESEARCH PAPER A: COST DRIVERS OF SOFTWARE CORRECTIVE MAINTENANCE: AN EMPIRICAL STUDY IN TWO COMPANIES

This research paper has been submitted to the International Conference on Software Maintenance (ICSM) 2010. It was accepted at June 29th 2010 for presentation at ICSM 2010 and to be published in the conference proceedings of ICSM 2010.

Cost Drivers of Software Corrective Maitenance: An Empirical Study in Two Companies

Jingyue Li, Tor Stålhane, Jan M. W. Kristiansen and Reidar Conradi
Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway
{jingyue, Tor.Stalhane, Conradi}@idi.ntnu.no

***Abstract*—To estimate the corrective software maintenance effort, we must know the factors that have the strongest influence on the effort of corrective maintenance activities. In this study, we have analyzed activities and effort of correcting 810 software defects in one Norwegian software company and 688 software defects in another. We compared the defect profiles according to the defect correction effort. We also analyzed defect descriptions and discussions between developers in the course of correcting defects to understand what led to the high cost of correcting some defects. The study shows that size and complexity of the software to be maintained, maintainers' experience, and tool support are the most influential cost drivers of corrective maintenance in one company, while domain knowledge is one of the main cost drivers of corrective maintenance in the other company. This illustrates that models for estimating software corrective maintenance effort have to be customized based on the defect profiles and cost drivers of each company and project to be useful.**

Keywords-software maintenance, maintenance effort, empirical studies

I. INTRODUCTION

Software maintenance can be costly. Some companies spent around 80% [1] or even more than 90% [2] of their software budget on software maintenance. It is therefore important to estimate the possible maintenance effort to facilitate planning and managing software maintenance.

Several studies have proposed methods to estimate the overall software maintenance effort [3-7], and effort of certain kinds of software maintenance, such as perfective maintenance [8], adaptive maintenance [9-10], or corrective maintenance [11-13]. Most of those studies first proposed or prepared candidate cost drivers of software maintenance, e.g. size of the system to be maintained [3], complexity of the system [5], or experience of the maintainers [4] by literature reviews [5], interviews with experienced maintainers [3, 14], or brain storming [15]. Those studies then chose the most influential cost drivers from the candidates and used them as parts of an estimation model, e.g. neural network, multiple regression, or pattern recognition, to estimate the maintenance effort. The candidate cost drivers occurring in the models varied widely. The most influential cost drivers were often chosen solely based on statistical data analysis, such as multiple regression analysis or principal component analysis. Few follow-up qualitative analyses have been performed to figure out why certain factors outperform others when it comes to effort estimation. Without qualitative analyses to interpret the influential cost drivers and models generated from such statistical methods, it is likely that the chosen cost drivers or models "over-fit" the data and therefore are biased.

To identify the most influential factors for corrective effort and their reasons, we have analyzed some corrective maintenance activities, i.e. defect fixing, of software systems of two large IT companies in Norway, named company A and company B. In both companies, we first introduced systematic defect classification schemes, such as IBM ODC [16], to classify their software defects into categories. Additionally, we have asked the developers to classify each defect based on the actual effort spent on fixing the defect. A few months after the defect classification schemes were deployed; we downloaded defect reports from these two companies and analyzed the defect reports quantitatively and qualitatively. The results from our study show that cost drivers of corrective maintenance are quite different in the two companies. The results indicate that companies must first analyze their own defect reports and corrective maintenance effort to identify appropriate cost drivers, and then choose an appropriate model to predict their corrective maintenance effort.

The rest of the paper is organized as follows. Section 2 gives an overview of previous studies on maintenance effort estimation. Section 3 presents the study design. Section 4 presents the results, and section 5 discusses our findings and possible limitations of our study. Section 6 concludes.

II. RELATED WORK

A. Cost drivers of software maintenance

Jørgensen [4, 17] showed that the maintenance effort was significantly influenced by the size of the maintenance task, the kinds of maintenance task (i.e.

corrective, perfective, or adaptive), and maintainers' confidence on the expected task, rather than which programming language were used, the priority of the task, the age and size of the system to be maintained, and the maintainers' experience. A later study by Jørgensen et al. [18] showed that experienced maintainers were better to estimate the effort of small and simple corrective maintenance tasks than inexperienced maintainers. Niessink and Vliet et al. [19] observed that the size of the software component to be modified had more impact on effort than the size of the change itself. Furthermore, they concluded that analogy and expert based estimation outperformed the function-points based estimation. To estimate maintenance effort, Anh et al. [5] used the function points of the system to be maintained and ten maintenance productivity factors, which were classified into categories, such as engineering skill, technical characteristics, and maintenance environment, Their results showed that the size of the system to be maintained was more influential than other factors on the maintenance effort [5]. Rao and Sarda [15] examined the cost drivers of software maintenance in a maintenance outsourcing context. They concluded that the five most influential factors were the multiple time zone support, size and complexity of the system, percentage of online system in the total system, and the nature of the service level agreement. Sarang and Sanglikar [6] examined the effort variances in several software maintenance projects and showed these were significantly influenced by the size of the maintained system, the size of the changes made, and the overall skill level of the maintenance team.

With respect to cost drivers of **corrective software maintenance**, Gorla et al. [12] used product metrics, e.g. size, complexity, and number of variables, as cost drivers to estimate the defect correction effort. They concluded that the use of GOTO, structures of the IF-ELSE constructor, and length of variable names determined the defect fixing effort. Evanco [20] concluded that defect locality, software complexity, number of subsystems, and the maintainers' familiarity with the software had the strongest impact on defect correction effort. However, the investigated cost drivers explained only 20% of effort needed to locate and fix the defects [20]. A later study by the same Evanco [21] showed that defects arising during system/acceptance testing were more costly to be fixed than defects discovered during unit testing. The reason for this was that defects discovered during system/acceptance testing might involve many subsystems and thus needed coordination among several engineers. In the study by De Lucia et al. [11], the size of the system to be corrected or modified was the key cost driver for the corrective maintenance effort.

Regarding the cost drivers for **adaptive software maintenance** tasks, Hayes et al. [10] showed that the number of lines of code changed and the number of operators changed had the highest correlation coefficient with the adaptive maintenance effort. Fioravanti and Nesi [9] also investigated metrics to estimate the adaptive maintenance effort. Their study showed that the metrics related to class complexity and size were the most suitable ones for predicting the adaptive maintenance effort.

The most influential cost drivers for software maintenance effort, based on the existing studies are summarized in Table 1, and show that different studies observed different cost drivers.

TABLE I. MOST INFLUENTICAL COST DRIVERS FOR SOFTWARE MAINTENANCE EFFORT FROM EXISTING STUDIES

Studies	The most influential cost drivers for software maintenance effort	Study context
[4, 17]	Size of the maintenance task Kind of changes Maintainers' confidence	Overall maintenance effort in a telecom. company
[18]	Maintainers' experience	Overall maintenance effort in a telecom. company
[19]	Size of the component to be modified	Overall maintenance effort of a financial information system
[5]	Size of the component to be modified	Overall maintenance effort of applications in various domains
[15]	Multiple time zone support Complexity of the system Percentage of online system in the total system Nature of the service level agreement	Overall maintenance effort in maintenance outsourcing context
[6]	Size of the maintained system Size of the changes made Overall skills of the maintenance team	Overall maintenance effort in two product lines within securities domain
[12]	GOTO usage Structure of the IF-ELSE construct Variable name length	Corrective maintenance effort of COBOL systems made by student
[20]	Defect locality Software complexity Familiarity with the software	Corrective maintenance effort of Ada projects in NASA
[21]	The phase where a defect was discovered	Corrective maintenance effort of Ada projects
[11]	Size of the system to be corrected	Corrective maintenance effort of software in various application domains
[10]	Lines of code changed Number of operators changed	Adaptive maintenance effort of two student projects and two research projects
[9]	Class complexity and size	Adaptive maintenance effort of an Object-Oriented music software system

B. Defect Classification Schemes

Defect classification schemes capture aspects of defects relating to when, where, and why a defect revealed itself. The most popularly used defect classification schemes are the IBM ODC [16], HP [22] scheme, and IEEE standard [23]. In IBM ODC [16], the attributes are organized into open and closed process steps. Attributes in the open steps are filled in when a defect has been detected, while closed steps attributes are filled in when the defect has been corrected and closed. An example

attribute of the open steps is activity (when a defect was detected, e.g. design inspection), while example attributes of the closed steps are type (what had to be fixed, e.g. algorithm) and qualifier (e.g. missing, incorrect, or extraneous).

III. RESAERCH DESIGN

The goal of this study is to investigate which cost drivers negatively impact the effort of software maintenance - especially of the corrective maintenance - and why. Unlike previous studies, the goal of the study is not to build an effort estimation model to help companies predict the possible maintenance effort, but to examine the reasons for high effort of corrective maintenance activities and thus to answer the question:

- *What can be done to reduce software corrective maintenance effort?*

The underlying objective of this study is software process improvement (SPI) in the national project EVISOFT [24], which includes eleven Norwegian IT companies. The study is also expected to investigate factors that impact software maintenance effort to validate or supplement observations from existing studies.

A. Investigated Companies and Projects

Company A is a software product line company with only one product. The product is deployed on more than 50 operating systems and hardware platforms. The company has more than 700 employees, including 400 developers and testers. The company is headquartered in Norway and branches in more than 10 other countries. The company releases a new version of its software every three months. The company has a Defect Tracking System (DTS), which records all the information of defects, from when they are reported till the defects are corrected and verified. Every month, thousands of new defects are registered in this DTS system. The development teams of this company are organized into three units. One unit (called core) is developing the reusable or core components to be used by all products of the product line. Another unit (called B2C) develops a shareware version of the product, where users can download the software for free, but without source code. The third unit (called B2B) is responsible for customizing, integrating, and packaging the product for commercial customers, who sell the product directly to end users.

Company B is a software house that builds business-critical systems, primarily for the bank and finance sector. The company has about 800 developers and testers and has branches in several big cities in Northern European countries. The company also has an internal Defect Tracking System (DTS), which is used by developers and tester to record and to track status of defects. Projects of this company are usually fixed-price projects, which are outsourced from financial companies.

B. Data preparation and data collection

As mentioned before, the main purpose of this study is to improve the software processes to increase testing effectiveness and to reduce defect fixing effort. Thus, defect classification schemes were introduced in both companies to classify defects and to enable effective defect analyses.

In company A, the DTS included several attributes for each defect, such as defect id, a short summary, detailed descriptions, the creation time of a defect report, reporter of the defect, estimated duration of the defect's correction, time when the defect correction is verified and closed, priority, severity, status, resolution, tester id, test description, version and hardware platform of the software with defects, and detailed comments and work log of the developers. Based on the IBM ODC [16] and the SPI goals of the company, its DTS was revised to include new or modified ODC defect classification attributes. The related ODC attributes to this study are shown in Table 2.

TABLE II. DTS SYSTEM EHNANCEMENT AND REVISION OF COMPANY A

Added or revised attributes	Value of the attributes
Effort to fix	Time-consuming: more than one person-day effort Easy-to-fix: less than one person-day effort
Qualifier	Missing; Incorrect; or Extraneous
Fixing type	Extended the ODC "type" attributes to reflect the actual defect correction activities
Root cause	Project entities, such as requirement, design, and documentation, which should be done better to prevent the defect earlier

In company B, the internal DTS system included attributes like defect id, a short summary, the time the defect is reported, reporter of the defect, priority, severity, defect status, tester id, and test description. The DTS of this company was also revised according to the company's SPI goals. Revised or added DTS attributes of this company that are related to this study are shown in Table 3.

In company A, six months after deploying the enhanced DTS, we downloaded information of defects, which were newly reported, confirmed as true defects, and had been corrected. 810 (40%) of these defects were chosen for later analysis, because they had the "effort to fix" attribute filled-in. In company B, six months after the enhanced DTS was deployed, we downloaded information of all 1032 defects from system tests in two releases of a large development project.

C. Data analysis

We used both quantitative methods and qualitative methods to analyze the defect information.

- First, we divided the defects into categories "easy-to-fix" and "time-consuming" based on the value of the "effort to fix" attributes assigned by

the developers in Company A. In company B, they had divided the defects into three categories based on the value of the attribute "effort to fix"

- Then, we compared the distribution of the values of the defect classification attributes, such as fixing type, root cause, and qualifiers for these two categories of defects in company A, and fixing type and root cause attributes in company B.
- We also read the discussions between developers during defect corrections in company A and the summary and test description of the defects in company B. Such information was analyzed using the cross-case analysis method [25].

TABLE III. DTS SYSTEM EHNANCEMENT AND REVISION OF COMPANY B

Added or revised attributes	Value of the attributes
Effort to fix	Simple : less than 20 minutes Medium: between 20 minutes and 4 hours Extensive: more than 4 hours
Fixing type	Attributes that reflect the actual defect correction activities
Root cause	Project entities, such as requirement, design, and documentation, which should be done better to prevent the defect earlier

IV. DATA ANALYSIS RESULTS

We hereby present the data analysis results of each company.

A. Data analysis results of company A

For the 810 defects of company A, their distribution based on the value of the attribute "effort to fix" is shown in Table 4 and show that 20% of them are costly to be fixed. Two causes led to few defects collected and analyzed from the B2B unit. One was that defects in this unit usually included the customer information and therefore blocked our access due to the non-disclose agreement between the company and its customers. Another reason was that projects of the B2B unit usually had a high time-to-market pressure. Thus, few developers would like to spent time filling in the defect classification attributes, because it was not mandatory to fill in these attributes in DTS.

The results of comparing several attributes of the "time-consuming" defects with those "easy-to-fix" ones are shown in Table 5. We see that:

- A significant more percentage of "time-consuming" defects were due to the weaknesses of the design phase than the "easy-to-fix" ones.
- At least 30% of the "time-consuming" defects in all units were due to missed functionalities.
- Different types of defect led to different defect correction efforts. For example, in the core and B2C units, significant more percentages of "time-consuming" defects were due to the "algorithm" type of defects (i.e. incorrect algorithms were implemented) than the "easy-to-fix" ones.

TABLE IV. DISTRIBUTION OF DEFECTS BASED ON THE VALUE OF THE ATTRIBUTE "EFFORT TO FIX" IN COMPANY A

Value of the attribute "effort to fix"	Number of defects with these attribute values	Business unit that the defects belong to
Time consuming	70	Core
	96	B2C
	6	B2B
Easy fix	222	Core
	408	B2C
	8	B2B

By analyzing discussions and conversations between developers in the course of correcting the "time-consuming" defects, we found several important reasons for the high effort of fixing these defects in company A. The number of defects of each cause is shown in Table 6. Each cause is shortly discussed below.

- **Hard to determine the location of the defect.** The product of the company has three millions lines of code. Additionally, the company is organized into three business units based on business focuses. When a defect was reported, it took time for developers to figure out the origin of the defect, because nobody had the complete picture of the necessary details of the product, due to its size.
- **Implemented functionality was new or needed a heavy rewrite**. Some defects were caused by missing or wrongly implemented functions, which did not satisfy the requirements specification. As such defects were usually introduced in the early development stages, such as requirement specifications and design, large segments of the software needed to be added or modified, which cost developers a lot of effort.
- **Long clarification (discussion) of defect.** When a defect was reported from a customer or tester, the developers used some time to discuss with its reporter to decide whether the newly reported defect was a real defect or a misuse of the product. There are a half million defect reports stored in the DTS. Thus, developers also had to clarify if a newly reported defect was a duplicate of a previously reported and fixed defect. Currently the DTS system of the company supports only keywords based search on the defect, which is not effective when it comes to identify defect duplications. For example, when a developer inputs the keyword "crash", thousands of defects will be returned.

TABLE V. COMPARSIONS OF OTHER DEFECT ATTRIBUTES OF THE "TIME-CONSUMING" DEFECTS AND "EASY-TO-FIX" ONES IN COMPANY A

Defect attribute	[a]Percentage of values of the defect attributes		Business unit
	Time-consuming defects	Easy-fix defects	
Root cause	14% in requirements phase 37% in design phase 49% in code phase 0% in build phase	16% in requirements phase 14% in design phase 67% in code phase 3% in build phase	Core
	13% in requirement phase 28% in design phase 57% in code phase 2% in build phase	8% in requirement phase 18% in design phase 72% in code phase 2% in build phase	B2C
	40% in requirement phase 60% in code phase	72% in requirement phase 28% in code phase	B2B
Qualifier	60% incorrect 3% irrelevant 37% missing functionality	55% incorrect 5% irrelevant 40% missing functionality	Core
	51% incorrect 0% irrelevant 49% missing functionality	45% incorrect 3% irrelevant 52% missing functionality	B2C
	25% incorrect 75% missing functionality	0% incorrect 100% missing functionality	B2B
Fixing type	38% algorithm 13% other fix 8% assignment / initialization 2% software interfaces 3% memory management 22% checking 3% function / class / object 5% timing /serial 6% standard compliance	17% algorithm 11% other fix 19% assignment / initialization 0% software interfaces 5% memory management 33% checking 5% function / class / object 1% timing /serial 9% standard compliance	Core
	39% algorithm 5% other fix 8% assignment / initialization 3% software interfaces 2% buffer / memory management 14% checking 23% function / class / object 6% timing /serial 0% standard compliance	15% algorithm 9% other fix 16% assignment / initialization 1% interface with 3rd party software 2% buffer / memory management 29% checking 25% function / class / object 2% timing /serial 1% standard compliance	B2C
	17% algorithm 17% other fix 32% assign / init 17% function / class / object 17% standard compliance	20% algorithm 20% other fix 20% interface with 3rd party software 20% checking 20% standard compliance	B2B

a. The missing data of each attribute were excluded when analyzing the percentage

- **The original fix introduces new defects / multiple fixes.** When developers think that they have fixed a defect, it is always a possibility that the new fix has introduced new defects elsewhere. Although regression testing is used to avoid such situations, this cannot be fully resolved, unless the test coverage of the regression testing is very high. In this company, some defects had to be reopened, reanalyzed, and fixed after the initial fix, which led to high effort for fixing such defects.

TABLE VI. CAUSES FOR HIGH EFFORT NEED TO FIX SOME DEFECS IN COMPANY A

Reasons for the time-consuming defect corrections	Numbers of defects related to each cause in each business unit		
	Core	B2C	B2B
Hard to determine the location of the defect	20	37	4
Implemented functionality was new or needed a heavy rewrite	13	29	2
Long clarification (discussion) of defect	19	5	0
The original fix introduces new defects / multiple fixes	13	9	0
Others (documentation is incorrect or communication is bad)	2	0	0
Reasons are not clear	3	16	0

B. Data analysis results of company B

The root cause attribute of the 1032 defects downloaded from this company has the following six categories:

- A: misuse or defect introduced by a sub-contractor.
- D: functional defect
- E: change request
- M: defect caused by wrong test data and in the system build process
- S: wrong or unclear specification
- U: defect introduced during development.

We ignored defects with categories A and E. In addition, we excluded 341 defects that had no value for "effort to fix" attribute filled in, thus reducing the number of defects for further analysis to 688. As the company categorized the correction effort into three categories, as shown in Table 3, we used the conservative values 1, 3 and 10 for these three categories respectively to calculate the cost index. As shown in Table 7 below, the majority of the defects with extensive correction costs stems from the development activities, i.e. category U. Data in Table 7 show that development (U) and requirements and design (S) account for almost 87% of all corrections costs. Using the fixing type attributes of the defects in the categories U

and S, we found that only four fixing types were needed to account for 70% of all correction efforts. These are Logical error (i.e. business logic of the system is wrong), Miscellaneous error (e.g. error message when trying to open some accounts), Wrong/unclear requirements (i.e. the requirement specification is wrong), and Wrong interface (i.e. the graphic user interface is wrong or meaningless information is shown to the user).

TABLE VII. ROOT CAUSES AND CORRECTION EFFORT FOR DEFCTS IN COMPANY B

Value of the "root cause" attribute	Value of the "effort to fix" attribute			Cost index	Percentage
	Simple	Medium	Extensive		
D	4	5	2	39	2,41
M	60	17	4	151	9,33
S	63	26	12	261	16,12
T	8	1	1	21	1,30
U	187	130	57	1147	70,85
Sum	322	179	76	1619	100,00

The "effort to fix" attribute of this company had three values. To make the data analysis of this company comparable with the analysis in company A, we merged defects of this company into two categories. We merge defects having "simple" and "medium" values of the "effort to fix" attribute of this company, because such defects cost less than 4 person-hours to be fixed. We als compared several attributes of the merged defects with defects having "extensive" value of the "effort to fix" attribute. The comparisons show that:

- A significant share of the defects fixed within four person-hours was due to mistakes in the graphic user interface, errors in the developer's test environment, or misinterpreted requirements.
- Logical defects accounted for a third of defects fixing with more than four person-hours.
- Defects due to missing functionality or weaknesses in the design phase lead to high defect correction effort.

Unlike company A, company B did not ask developers to insert information on discussions among themselves in the DTS system. However, by qualitatively analyzing the defect summary and test descriptions, we could still get an insight into why the defects were introduced and the reasons for the high effort needed for defect fixing. The qualitative analyses show that approximately 95% of defects were introduced due to insufficient domain knowledge of the developers. This led to wrong/missing functionalities or wrong/missing information displayed. The company hired many external consultants, who had excellent programming skills and knowledge. However, their knowledge of the financial application domain was limited.

V. DISCUSSION

A. *Cost drivers of corrective software maintenance effort*

Jørgensen [4] showed that productivity of corrective maintenance was significantly different from that of the adaptive and perfective maintenance. Although Henry and Cain [8] showed that the differences of productivity of perfective and corrective maintenance activities were not significant, their results illustrated that the variance of productivity of the corrective maintenance was significantly higher than the productivity variance of the perfective maintenance activities. De Lucia et al. [11] also showed that different kinds of corrective maintenance tasks, i.e. tasks requiring software modification, tasks requiring fixing of data misalignment, and other tasks needed different effort estimation models.

The results of our studies have contributed to explain why the variances of corrective maintenance can be high, and why different effort estimation models have to be explored for different corrective maintenance tasks. Our data show that several factors can be important cost drivers for corrective maintenance, such as:

- **Size of the system to be maintained**, as discovered in [5] [6] [11] [19]. The larger the size of the system, the more difficult for developers to have a complete picture of the system and to be able to locate and eliminate the defects quickly. Therefore, a lot of effort may be spent on discussing where a defect stems from.
- **Complexity of the system to be maintained**, as illustrated in [9] [15] [20]. Although we did not measure the complexity of the investigated systems, data show in Table 6 showed that a complex architecture of the maintained system could impact the maintenance effort. As system size, high system complexity also makes it difficult to understand the system, and to locate and correct the defects. In addition, fixing defects in a complex system may easily introduce new defects. Jones [26] showed that defect removal efficiency of defects introduced due to bad fixes was only 70%. Xie and Yang [13] also proposed that a cost model should include the possible extra cost from imperfect debugging.
- **The phase a defect is discovered**, as mentioned in [21]. Our data show that when the defects were introduced in the early lifecycle stages and discovered in system or acceptance testing, many lines of code have to be added or modified. The heavy changes of the system will lead to high corrective maintenance effort.
- **Maintainers' experience with maintained system and the application domain of the system,** as discovered in [6] [18]. Data in Table 6 show that familiarity with the system itself can reduce the time spent on discussing and locating

the defects. Analyses of defects in company B show that sufficient domain knowledge can help developers understand the defects quickly and thus improve the defect fixing efficiency.

- **Tool and process support.** Results from company A show that many possible duplications of the defects led to a waste of developers' time. Results from company B show that the external consultants' lack of domain knowledge slowed down both software development and maintenance.

B. Generaliability of the model for estimating maintenance effort

In study [3], seven candidate cost drivers of software maintenance were extracted from one company and 10 candidate cost drivers of software maintenance from another company through literature reviews and interviews with managers and engineers. Among those candidate cost drivers, only three of them were mentioned in both companies. After a principal component analysis, the candidate cost drivers in both companies were clustered into three similar factors. However, those factors did not explain the variance of effort in one of the two investigated company well in a regression analysis.

Our study shows that a corrective maintenance effort model has to be heavily customized from company context. Company A is a product line company, which has a very large system, a complex architecture, and a DTS system with many defect reports. The cost drivers of maintenance of this company are size, complexity, maintainers' experience, and tool support. Company B works in several independents projects, each having a smaller size than the single system in company A. The main cost driver of projects we investigated in company B is the insufficient domain knowledge. The domain knowledge is not an issue in Company A, as all its developers are internal ones and having a solid understanding of the application domain.

C. Recommendations for industry

De Lucia et al. [11] examined defect rates and the mean time between defects to help estimate the corrective maintenance effort. We also recommend industrial practitioners to investigate their companies' defect reports and find out their own cost drivers for corrective maintenance, before using any effort estimation model to estimate their maintenance effort. A company should randomly choose some defect reports and:

- Classify the defects into orthogonal categories.
- Analyze the defect rate and mean time between defects of each category of defects.
- Analyze the cost drivers and defect fixing productivity of each category of defects.
- Use the defect profiles of a project and the mean effort of fixing its defects to estimate the corrective maintenance effort of similar projects.

However, industrial practitioners must be aware that cost drivers of corrective maintenance are dynamic, not static. For example, in company A, once an efficient tool to identify possible defect duplication is deployed, the impact of defect duplication on the maintenance effort will be reduced. In company B, once domain knowledge of the external consultant is improved, the weight of factor "insufficient domain knowledge" in an estimation model should be reduced.

D. Recommendations for academia

Our results show that the cost drivers of different companies are different. The cost drivers can also be different in different projects of the same company. Thus, for any proposed models to estimate the maintenance effort, we recommend researchers to cross-validate the models in different contexts, or at least explicitly report the context of the company and project used for validation in sufficient details to generalize the model properly.

E. Possible threats to validity of the study

1) Possible threats to internal validity: One threat to internal validity of the study is that developers might classify the defects wrongly and therefore biased our data analysis results. The threat is eliminated by our qualitative analysis, i.e. by reading the detailed log of the defect fixing in company A and by reading the description of defects in company B. We have corrected a few defects, which were obviously wrongly classified. For example, some defects were initially classified as "easy-to-fix" in company A. However these defects were later re-opened and fixed in a "time-consuming" way due to the initial unsatisfactory fixing. We changed the classification of such defects into "time-consuming" in our data analysis.

2) Possible threats to conclusion validity: One threat to conclusion validity of the study is the missing data of certain defect classification attributes. In Table 5, we excluded those missing data in data analysis, which might bias our results. However, such a missing data problem is difficult to be avoided in real industrial studies, especially when the data to be filled in are not mandatory.

3) Possible threats to external validity: Our results show that cost drivers of maintenance effort can be very different across companies and projects. Thus, generalizing results of our investigated companies should be done with caution.

VI. CONCLUSIONS AND FUTURE WORK

Several cost drivers and models have been proposed for estimating software maintenance effort. This study investigated around 1500 defect reports from two companies in details. First, we introduced comprehensive defect classification schemes in the investigated companies. Then, developers assigned values to the defect classification attributes during defect fixing. By analyzing

the defect attributes and detailed information of the defects, especially focusing on the differences between the defects that were fixed quickly and those that were fixed slowly, we observed that important cost drivers of maintenance effort of the two companies are different, due to different company and project contexts. The results from our study illustrate that models used to estimate software corrective maintenance effort have to be customized and updated regularly based on analyses of the defect profiles and cost drivers of defect corrections.

As the companies involved in our study will improve their corrective maintenance efficiency after seeing results of this study, a future study is to examine the impact of certain software corrective maintenance improvements, for example introducing a better defect duplication detection tool in company A, to facilitate decision makings of implementing these improvements.

ACKNOWLEDGMENT

This study was supported by the Research Council of Norway through the project EVISOFT (174390/I40).

REFERENCES

[1] T.M. Pigoski, *Practical Software Maintenance: Best Practices for Managing Your Software Investment (1st edition)*, Wiley, 1996.

[2] E. Len, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, 2000, pp. 17-23.

[3] F. Niessink, "Two case studies in measuring software maintenance effort," *Proc. IEEE International Conference on Software Maintenance (ICSM'98)*, IEEE Computer Society, 1998, pp. 76.

[4] M. Jørgensen, "Experience with the accuracy of software maintenance task effort prediction models," *IEEE Transactions on Software Engineering*, vol. 21, no. 8, 1995, pp. 674-681.

[5] Y. Ahn, et al., "The software maintenance project effort estimation model based on function points," *Journal of Software Maintenance*, vol. 15, no. 2, 2003, pp. 71-85.

[6] N. Sarang and M.A. Sanglikar, "An analysis of effort variance in software maintenance projects," *Advances in Computer and Information Sciences and Engineering*, T. Sobh, ed., Springer Netherlands, 2008, pp. 366-371.

[7] R. Shukla and A.K. Misra, "Estimating software maintenance effort: a neural network approach," *Proc. 1st India software engineering conference*, ACM, 2008, pp. 107-112.

[8] J.E. Henry and J.P. Cain, "A quantitative comparison of perfective and corrective software maintenance," *Journal of Software Maintenance*, vol. 9, no. 5, 1997, pp. 281-297.

[9] F. Fioravanti and P. Nesi, "Estimation and prediction metrics for adaptive maintenance effort of object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, 2001, pp. 1062-1084.

[10] J.H. Hayes, et al., "A metrics-based software maintenance effort model," *Proc. Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, IEEE Computer Society, 2004, pp. 254.

[11] A. De Lucia, et al., "Assessing effort estimation models for corrective maintenance through empirical studies," *Information and Software Technology*, vol. 47, no. 1, 2005, pp. 3-15.

[12] N. Gorla, et al., "Debugging effort estimation using software Metrics," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, 1990, pp. 223-231.

[13] M. Xie and B. Yang, "A Study of the effect of imperfect debugging on software development cost," *IEEE Transactions on Software Engineering*, vol. 29, no. 5, 2003, pp. 471-473.

[14] M. Jørgensen and D.I.K. Sjøberg, "Impact of experience on maintenance skills," *Journal of Software Maintenance*, vol. 14, no. 2, 2002, pp. 123-146.

[15] B.S. Rao and N.L. Sarda, "Effort drivers in maintenance outsourcing - an experiment using taguchi's methodology," *Proc. Seventh European Conference on Software Maintenance and Reengineering*, IEEE Computer Society, 2003, pp. 271.

[16] R. Chillarege, et al., "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, 1992, pp. 943-956.

[17] M. Jørgensen, "An empirical study of software maintenance tasks," *Journal of Software Maintenance*, vol. 7, no. 1, 1995, pp. 27-48.

[18] M. Jørgensen, et al., "The prediction ability of experienced software maintainers," *Proc. Conference on Software Maintenance and Reengineering (CSMR'00)*, IEEE Computer Society, 2000, pp. 93.

[19] F. Niessink and H.v. Vliet, "Predicting maintenance effort with function points," *Proc. International Conference on Software Maintenance (ICSM'97)*, IEEE Computer Society, 1997, pp. 32-39.

[20] W.M. Evanco, "Modeling the effort to correct faults," *Journal of Systems and Software*, vol. 29, no. 1, 1995, pp. 75-84.

[21] W.M. Evanco, "Prediction models for software fault correction effort," *Proc. Fifth European Conference on Software Maintenance and Reengineering*, IEEE Computer Society, 2001, pp. 114-114.

[22] R.B. Gardy, *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992.

[23] IEEE, "*Standard Classification for Software Anomalies - IEEE Std. 1044-1993*", 1994.

[24] "EVISOFT project," http://geomelding.geomatikk.no/evisoft/.

[25] A.C. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory (2nd edition)*, Sage Publications, 1998.

[26] C. Jones, *Software Quality: Analysis and Guidelines for Success*, International Thomson Computer Press, 2000.

APPENDIX B

DRAFT OF RESEARCH PAPER B: ENHANCING SOFTWARE DEFECT TRACKING SYSTEM TO FACILITATE CONTINUOUS SOFTWARE QUALITY ASSESSMENT AND IMPROVEMENT

At the time of writing, this research paper has been submitted to IEEE Software and is currently under review.

Enhancing Software Defect Tracking System to Facilitate Continuous Software Quality Assessment and Improvement

Data in a defect tracking system is one of the most important sources of information for software quality improvement. Most previous studies that utilized software defect data did their work after the actual project was finished. However, for projects that rely on empirical control of processes and that deliver working versions of software frequently, for example, by using agile methods, quality managers need to examine problems that pertain to software quality weekly or monthly. It is therefore critical that the defect tracking system can provide timely, relevant, reliable, and easy-to- analyze data. We investigated defect tracking systems of nine Norwegian companies. We improved the systems in two of these companies by introducing new defect classification attributes or customizing existing ones. Experience gained and lessons learned from the studies provided valuable insights into benefits of and barriers to improve an existing defect tracking system for continuous software quality assessment and improvement.

Keywords:

D.2.19.d Measurement applied to SQA and V&V, D.2.18.g Process implementation and change

1. Introduction

Following ISO9001 [1], section 7.2.3 on customer communication, most software companies have **a Defect Tracking System (DTS)**. Although customers, developers, and testers use the DTS mainly for reporting defects and for tracking defect fixing status, they can also use the defect data to assess the quality of software and to identify weaknesses in software development procedures and testing. However, most studies that utilized the defect data for software quality assessment, such as [2], have been retrospective, i.e. performed by researchers or dedicated quality managers after a project was finished. Making improvements only after a project is finished is often too late, especially for software processes that rely on the empirical control of processes and deliver versions of working software frequently. If developers and testers are asked to identify the root causes of failures that occurred several months before, the data they provide might not be reliable. In addition, without a DTS that is designed specifically for Software Quality Assessment (SQA) and Software Process Improvement (SPI), developers and testers may not be aware of or motivated to provide the data that is needed for later defect analyses. We examined the DTS of nine Norwegian companies and found that most of the data collected in these systems were either never used, or were irrelevant, unreliable, and/or difficult to use for SQA and SPI.

In response to our findings, we developed improved versions of the DTS in two companies. Our purpose was to enable decisions about SQA and SPI to be made as early as possible. Our main focus was to either (a) introduce new defect classification attributes into existing DTS or (b) revise existing defect attributes,

so that developers and testers can better provide relevant, reliable, easy-to-analyze data on defects for process analyses in a cost-effective manner.

A few months after the companies had implemented the improved DTS; we downloaded data about the newly reported defects and analyzed them. The defect analyses yielded valuable insights into the weaknesses of the companies with respect to SQA that could not have been acquired without the improved DTS. In addition, in one company, we held a workshop to collect feedback on the improved DTS. In the other company, we conducted a survey by email for the same purpose. The collected feedback shows that to ensure the reliability and completeness of the reported defect data, several organizational factors should be addressed.

2. Software quality assessment and improvement

Software quality can be described from several viewpoints [3], for example:

- **Users**: how well does the software product meet end users' concrete needs?
- **Manufacturing**: how well is the software developed the first time?
- **Product**: how good is the software quality according to internal quality indicators, such as code complexity?
- **Value-based**: how cost-effective are the processes for software development, testing, and defect fixing?

As mentioned above, companies can use data on software defects to track and improve the software development and testing processes. Several schemes, such as IBM ODC scheme [4], Hewlett Packard scheme [5], and IEEE standard scheme [6], for classifying software defects have been proposed to analyze defect data for SQA and SPI purposes. For a defect classification scheme to be useful, it is important to tailor the scheme to the company's needs [7].

3. Defect data in our investigated DTS

Attributes included in the DTS of the nine companies that we have examined are marked by "X" and shown in Table 1. We see that:

- All companies used free text for defect summary, description, or comments items. Most companies recorded the location, priority or severity, and calendar dates of reporting and resolving defects.
- None of the companies recorded the effort used to fix defects. Only two companies estimated the duration or effort needed to fix defects.

- Only two companies recorded how defects were fixed. Most companies were satisfied that a defect was fixed, without caring about how.
- Several companies, such as companies AN, CS, SN, and DA, did not use a separate item to describe how defects were discovered. The procedures that the employees used to discover defects are mixed with other text in the short summary or detailed description of defects.

Referring to software quality viewpoints [3] and the attributes of the defect classification schemes in [4][5][6], we found that some defect data of our examined companies can be used for SQA and SPI purposes:

- Six of the nine companies used a dedicated item to record the email address or name of the person who created the defect report. It is therefore possible to distinguish between defects that internal testers reported and those that external customers reported. Combining this information with defect severity, companies can see how many severe defects the testing team failed to detect. The greater the number of severe defects that the customer reports, the lower the customer satisfaction is likely to be.
- Seven companies recorded the name of the infected "module". This information can help developers to identify the most error-prone parts of the system, which are obvious targets for quality improvement.

Although a great deal of information related to SQA and SPI was available, none of the companies had used the information for such purposes. The assembled information was mainly a data graveyard. Furthermore, close investigation of defect data illustrated that some data were missing, and even the data recorded were difficult to analyze or too unreliable to be used for SQA and SPI.

- **Difficult to analyze**: The free text in comments and work logs includes a large and diverse amount of information. For example, developers' complaints about the complex architecture during defect fixing indicate that software quality is poor from the manufacturing viewpoint. Information in the work logs indicate what should be improved to speed up the defect fixing. However, without costly manual investigation, and/or customized text-mining tools, it is difficult to extract and interpret the information that is needed for analysis.
- **Missing and inconsistent data**: Although developers or testers are required to provide complete and accurate data, a lot of data are incomplete, because the company process allowed closing a defect without providing information for all attributes. When the definitions of the attributes are vague, the data that is provided will often be inconsistent. For example, we found that some people used the name of an embedding module or subsystem as the location of a defect, while others gave the name of a function.

Table 1. Defect attributes in the examined DTS

Items in the DTS		Company name								
		AN	CO	CS	PW	DP	SN	DT	SA	DA
		Total number of employees of the company								
		320	180	92.000	500	6.000	400	9.000	30.000	10
Descriptions	Id	X	X		X	X	X	X	X	
	Short summary				X	X	X	X	X	
	Detailed description	X	X	X	X			X		X
	High-level category*	X	X	X	X		X	X	X	
Timestamp and assignee	Create time	X	X	X	X	X	X	X	X	X
	Create by	X	X	X	X		X	X	X	
	Modified time	X	X	X	X	X		X	X	
	Modified by	X				X	X			
	Responsible person		X	X	X		X	X		
	Due time				X		X			
	Closed time	X			X					X
	Estimated duration to fix				X				X	
	Remaining time to fix				X				X	
Impact identification	Priority	X	X		X	X	X	X	X	
	Severity	X	X	X	X	X		X	X	
Fix	Status	X	X		X	X	X	X		
	Resolution				X		X			
	Release solved	X			X					
Test activity	Test by					X		X	X	
	Test ID		X		X	X				
	Test priority		X							
	Test description		X		X	X		X	X	
Location	Release	X			X					
	Module	X	X	X	X		X	X	X	
	Version		X	X	X		X			
	Operating system				X					
	Hardware platform				X					
Supplementary info.	Comments	X	X		X			X		X
	Related link				X		X	X		
	Work log			X	X				X	

* High-level category: bug/enhancement/duplication/not-bug

- **Lack of necessary data**: As shown in Table 1, few companies recorded the actual effort spent on fixing defects. Further, they collected only limited information that would enable the assessment of software quality from a value-based view.

4. Two case studies of improving DTS

We improved the DTS of two companies, namely DP and PW in Table 1, by following the process of reusing and introducing a defect classification scheme proposed by the IEEE standard [6].

4.1. DTS improvement in company DP

Motivation: Company DP is a software house that builds business-critical systems, primarily for the financial sector. Personnel in different departments of the company used the existing DTS in different ways. Due to the fact that nobody used the system for anything that the developers considered useful, there were few incentives to improve either the system or its use. However, a gap analysis that we performed in this company showed that one of the main concerns of the testers and developers is the company's defect reporting and prioritization process. Another main concern is to reduce the defect fixing effort.

Goal: We set ourselves the task of improving the DTS to provide supplementary information that the Quality Assurance (QA) managers could use to assess their correction costs and to answer questions such as the following:

- What were the main types of defect?
- How much effort did developers spend on defect fixing?
- What can developers do to avoid defects in the early stages of a project?

Upgrade proposal: We based our proposal for improving the system on an analysis of existing data and suggestions that the company's QA manager provided. To avoid abrupt changes, we did not introduce new attributes, only revised the categories for the existing attributes.

Validation: We performed two rounds of validation of the proposal for improvement, together with the test manager, one developer, and one project manager, through a detailed analysis of defects from earlier projects. The improved attributes that we made to the DTS of this company, after validations are:

- Fixing type: Introduced a new set of attributes to reflect the defect fixing activities of the developers.

- Effort: Three values that classify effort to reproduce and fix defects in “simple”, “medium”, and “extensive”. “Simple” means that the developers spent less than 20 minutes of effort. “Medium” means effort that the developer spent was between 20 minutes and 4 hours, and “extensive” means the effort that developer spent was more than 4 hours.

- Root cause: The attributes here are project entities, such as requirement, design, development, and documentation, the improvement of which will enable employees to prevent defects from occurring.

Follow-up: After the validation, we gave a presentation to developers, testers and project managers, in which we explained what the company could use the revised attributes for and why this was important for them. We also revised the work flow of the system, so that developers and testers had to provide all required attributes except “effort” in order to be able to close a defect.

4.2. DTS improvement in company PW

Motivation: Company PW is a software product line company, with only one product, but they deploy this product on more than 50 different operating systems and hardware platforms. The results of a similar gap analysis in this company show that their QA persons prioritized a more formal DTS. The QA manager wanted a mechanism that would allow them to analyze defect information quickly and continuously, because the company receives thousands of defect reports every month.

Goal: To improve the DTS to provide supplementary information that the company could use to assess software quality from a manufacturing and a value-based view and to answer questions such as the following:

- Which testing activities discovered or reproduced most defects?

- Why did developers spend on so much time on defect fixing?

- What can the company do to prevent defects in the early stages of a project and to detect defects before the software reaches the customers?

Upgrade proposal: We added and revised attributes of the defects of the system based on the IBM ODC [4], the “suspected cause” attribute of the IEEE standard [6], and suggestions from the QA managers of the company.

Validation: We performed two rounds of validation, together with one company QA manager, one tester, and one developer, through classifying defects that had occurred in previous projects. The newly added and revised attributes that we made to the DTS of this company, after validations, are:

- Effort: Two values that classify effort to reproduce and fix defects in “quick-fix” and “time-consuming”. “Time-consuming” means that the developers spent more than one person-day of effort.
- Qualifier: Attributes are “missing”, “incorrect”, and “extraneous”.
- Fixing type: Extended the IBM ODC “type” attributes with possible defect fixing activities of the company.
- Severity: The values of the attributes are defined on the basis of the effect of the defect on running the software, for example, crash the operating system, crash the software, significant, and trivial.
- Trigger: Classify testing methods on the basis of the company’s testing activities of the product.
- Root cause: The attributes here are project entities, such as requirement, design, and documentation, the improvement of which will enable employees to prevent defects from occurring.

Follow-up: After the validation, we gave a presentation to project managers to explain the added or revised attributes. We uploaded online manual to the DTS to help people to use the improved DTS. To avoid making large changes of the system, we separated the newly added attributes from the existing ones as “extra” attributes.

5. Software quality insights in the companies from improved DTS

5.1. Company DP: Defect data supplemented the results that Post Mortem Analysis (PMA) yielded

In company DP, six months after the company had deployed the new version of the DTS; we downloaded information on 1053 defects from system tests in two releases of a large system. By analyzing the root cause and fixing type attributes, we found that 397 of these defects were related to development and were responsible for 71% of defect fixing effort. We found that the majority of these 397 defects comprised wrong/missing functionality or incorrect/missing information. When the QA manager saw the results of this data analysis, she realized that those defects were caused by hiring a large number of consultants who had excellent development and coding experience but insufficient domain knowledge of banking systems. When the developers did not possess sufficient domain knowledge of the intended application area, system quality suffered and many defects related to incomplete functionality or incorrect business rationality occurred. Without such defect data and analysis, we would not have acquired this insight, especially when an early PMA showed that developers

of this company were proud of their application knowledge and preferred high-level requirements specifications, because this allowed them to use their creativity in later design and coding.

In retrospect, we see that an early and continuous analysis of defect reports would have revealed that the project was lacking application domain knowledge. The company could have prevented this either by adding domain experts to the project or by writing clearer and self-contained requirements.

5.2. Company PW: Defect data helped to support project manager SPI decision

In company PW, six months after the company had deployed the improved DTS; we downloaded and analyzed 796 defects from two projects. The developers classified 166 of these defects as time-consuming. Simple statistical analyses of the 796 defects show that:

- 60% of the defects that the developers classified as "time-consuming" were related to defects that can be easily detected by code reviews, such as wrong algorithms, or missing exception checking and handling.
- 20% more of the root causes for "time-consuming" defects than for the "quick-fix" defects were related to bad design. In addition, 40% of the values for the "qualifier" attributes of the "time-consuming" defects were "missing". This finding indicates that a lot of the defect fixing efforts for such defects comprised new implementation, because the sought function was absent from the initial design.

One project manager of the two examined projects had a feeling that his project needed code and design reviews. However, he had no solid data to support his feeling to make a decision. In retrospect, the defect analyses revealed that this project could have prevented many defects and saved defect fixing effort by introducing more formal code and design reviews.

6. Discussion

Our defect analyses in these two companies and the feedback that we collected from the evaluation activities illustrate important issues that companies need to take into account when improving DTS, so that they can support continuous and more agile SQA and SPI.

6.1. Cost-effectiveness of the improved DTS

Effort of providing defect data will be compensated for by the subsequent occurrence of fewer defects. Although developers and testers need to expend a little more effort on providing data in the improved DTS, these efforts pay dividends. Providing information about a defect takes only person-seconds. Fixing a defect may take several person-days. For example, in company PW, the ratio of the number of "time-

consuming" to "quick-fix" defects is 1:4. Avoiding just a few "time-consuming" defects in the early stages of a project, by utilizing data from the DTS, will more than offset the extra effort spent on providing defect data.

Our experience from company DP shows that developers are not against collecting data *per se*; rather, they are against collecting data when they do not see any concrete benefits for them. Thus, personnel who are supposed to provide more defect data need to be shown that the company will use the data for something that will benefit them in their day-to-day work. For agile development, defect data that the improved DTS provides can help quality managers identify weaknesses in software quality and development processes right after an iteration, so that they can make decisions about improvement early and continuously. The fast response to the defect data that developers provide will show them the actual benefits of providing such data and motivate them to put sufficient effort into supplying it.

6.2. How to ensure quality and completeness of the data?

Training is critical. Abstract category names are found to be a problem when using defect classification schemes [8]. Some developers of company PW complained, during the evaluation of our improved DTS, that they did not fully understand the revised classifications, because the training of the improved DTS was only performed within project managers. Thus, all personnel who are involved need to understand and accept all categories that a DTS uses. That is, they need to be able to see the categories as being relevant, they need to see the work as doable, and they must have ready access to a manual or online help system.

Company level validation is necessary. Improving a DTS is an organizational-level task. In company PW, although two senior members from the unit that serves commercial customers cross-checked the proposal for improvement, comments from the evaluation showed that the improvements were suitable mostly for this particular unit and not for the whole company. However, the classification would be too complex if those responsible for the improvement were to include all company's testing and defect fixing activities in the domain of attribute values. Therefore, there has to be a tradeoff between completeness and local suitability.

Make the right person provide the right data at the right time. A defect goes through several states in company PW: from newly reported, confirmed by testers, to resolved and verified. In our improved DTS, we ask the testers provide information about the attributes about which they have knowledge, such as "trigger", when a defect is reproduced and confirmed. We ask developers to classify effort on the basis of their actual fix of the defect, when the defect is completely resolved and verified. However, without support from proper work flow, sometimes information is not being provided and sometimes the information that is provided is inconsistent. For example, we found that the "trigger" attributes were rarely provided, because testers were not forced or reminded to provide them. From reading the work

log, we also found that some defects, whose "effort" attribute were classified as "quick-fix", should actually have been classified as "time-consuming". This misclassification occurred in the following manner: developers initially classified such defects as "quick-fix" after a short defect fixing session; then, the defect was later reopened and re-fixed in a "time-consuming" way, but not reclassified. In company DP, we were able to revise the workflow of the DTS to remind or even force testers and developers to provide certain defect data before they close a defect report. Therefore, missing or inconsistent data in this company is rare.

Default values can severely bias results. Values of some of the attributes (e.g. severity) in the improved DTS of company PW were placed in a drop-down menu with a default value to illustrate the meaning of the attribute. The results of our analysis show that more than 70% of the defects have the default value. By reading the work logs of many such defects, we found that some of these should have a different severity level. We suspect that developers and testers simply skipped this attribute, when they saw that a default value was already provided.

An orthogonal classification scheme does not mean that only a single choice is available during defect classification. We found that the allowed attribute values should sometimes be multiple-choice to be more applicable, as researchers also discovered in [8]. For example, the fixing of a complex defect may include correcting the assignment of a variable and the algorithm for using the variable. Classifying fixing type of such a defect as either an "assignment" or "algorithm" only will be incorrect.

The company should update the values of the defect attributes regularly. When the company's practices change, the values of the defect attributes have to follow suit. In company PW, the developers started perform code and design reviews after we deployed the improved DTS and showed the preliminary defect analysis results. Thus, DTS of this company should be updated to include the code review practice in the "trigger" attributes, so that developers can classify such an activity.

6.3. How can the company facilitate continuous defect analysis?

The improvement of a DTS should be goal-oriented. Before proposing an improvement to the system, company managers should have a clear idea of what analyses they want to perform and why they want to perform them. Statistical analysis tools should be part of the DTS to generate appropriate defect analyses easily and timely.

Data analysis should combine several attributes. To get a complete picture of defects, it is important to combine data values for several attributes. In company PW, when we analyzed defect fixing activities, we combined the "qualifier" attributes with the "root cause" attributes. We found that many costly bug-fixing sessions were caused by missed functions from the design phase. Given that fixing such a bug is

similar to new development, it was unsurprising that employees had spent a great deal of effort on them.

7. Conclusion

Although we improved the DTS in the two companies DP and PW, we found that two issues warrant further investigation.

How should the company collect and analyze data to identify the root causes of a defect reliably? In company PW, we let developers or testers specify the root causes of each defect from different aspects, such as wrong requirements definition or bad design. However, it turned out that developers knew only what was happening in the code without being able to trace the causes of a defect back to earlier stages of a project. Thus, the root cause attribute should probably be specified by different persons from different perspectives; but by whom, and how can personnel resolve conflicting proposals? In company DP, the statistical analysis of the defect data said a lot about what was occurring, but provided limited information about why it was occurring. In this company, we had to identify the root causes of defects by combining statistical analysis of defect data with information in the defect description (free text) and knowledge of QA managers. Thus, further research needs to investigate how to facilitate root cause analysis with less costly human involvement.

How can the company change and develop the defect scheme without affecting existing data? As we observed in company PW, the company must develop the defect classification scheme according to new or changed development and testing practices. However, such classification scheme changes may invalidate previous provided data and make it impossible to compare new and old data.

8. References

[1] ISO 9001:2000, “Quality management system – Requirements”, third edition, 2000-12-15.

[2] M. Butcher, H. Munro, and T. Kratschmer, “Improving Software Testing via ODC: Three Case Studies,” *IBM Systems Journal*, vol. 41, no. 1, 2002, pp. 31-44.

[3] B. Kitchenham and S. L. Pfleeger, “Software Quality: the Elusive Target,” *Software*, vol. 13, no. 1, 1996, pp. 287-296.

[4] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 1, 1992, pp. 943-956.

[5] R. B. Grady, "Practical Software Metrics for Project Management and Process Improvement," Prentice Hall, 1992.

[6] IEEE Standard Classification for Software Anomalies, IEEE Std. 1044-1993, 1994.

[7] B. Freimut, C. Denger, and M. Ketterer, "An Industrial Case Study of Implementing and Validating Defect Classification for Process Improvement and Quality Management," *Proc. 11th IEEE Metrics Symposium*, 2005, pp. 19.

[8] A. A. Shenvi, "Defect Prevention with Orthogonal Defect Classification," *Proc. of the India software engineering conference,* 2009, ACM press, pp. 83–88.

APPENDIX C

PRESENTATION SLIDES FROM MEETING AT COMPANY X 23TH MARCH 2010

The following slides were developed to the meeting with Company X on the 23th March 2010. The slides were created by J. M. Kristiansen, Jingyue Li and had contributions from Reidar Conradi. The slides have the name of Company X replaced with "Company X", and representatives from Company X have been given pseudonyms like "Person A".

Company X and EVISOFT/NTNU

SPI research with gap- and defect analysis, 2008-2010

Manager feedback meeting

23 March 2010

Reidar Conradi, Tor Stålhane,

Jingyue Li, Jan M. W. Kristiansen - NTNU

Outline

- IDI department at NTNU
- Background of EVISOFT and Company X
- Plan and status of the studies
- Results of two studies
  - S1: Gap analysis (2008)
  - S2: Defect analysis by improving the Bug Tracking System. BTS (2009-2010)
- Recommendations for future studies
- Discussion

2

IDI Department at NTNU

- 140 employees – 31 nationalities!
  - 45+ teachers (faculty w/ six women), 22 tech./adm., 52 PhD fellows, 20 temporary researchers/postdocs/teachers incl. 8 adjunct teachers (II"ere).
- 800 full-time students
  - participating in 7 study programs
  - 6000 individual exams per year
  - 150 master candidates per year
  - 10 PhD candidates per year.
- Annual budget (2008):
  - 67 MNOK from NTNU
  - 18 MNOK from projects.
- Important **value chain**: teachers – postdocs – PhD students – master students – bachelor students – IT industry.

3

IDI Department at NTNU (cont')

- 166 counting "Frida" publications in 2007, 159?? in 2008, ?? in 2009.
- 11 research groups
- Software engineering (SE) group in 2009:
  - 5,2 teachers, 2 researchers, 17 PhD fellows – 13 nationalities!
  - Papers: 44/61 of 166/196 (2007), xx/47 of 1xx/181 (2008), xx/nn of 1xx/163 (2009);
    i.e. **25 % of IDI total**, ca. 20 each year w/ foreign colleagues; 500 papers in last 10 years.
  - Ca. 25 master candidates/year; 3 PhDs in 2009, 4? In 2010.
  - 7 MNOK in external projects (40% of IDI total.)

6/11/2010 Manager feedback meeting 4

Context

- EVISOFT
  - User-driven research project financed by Norwegian Research Council (2006-2010), 8.5 mill. NOK in 2010.
  - Focus: Experience-driven Software Process Improvement (SPI)
  - 3 research partners: SINTEF, NTNU, and UiO
  - Ten IT companies in Norway: ABB, DNV Software, EDB, Firm, Geomatikk IT (coord.), Kongsberg Spacetec, KnowIT, Software Innovation, Telenor, and Vital.
- EVISOFT and Company X software
  - Company X Software became the 11th industry partner in Oct. 2007, cooperating with NTNU researchers.
  - Company X part formally started in Nov. 2007.

5

Goals (EVISOFT plan document)

- Company X Software's business goals
  - Increase effectiveness and efficiency of software testing
  - Increase productivity via reuse in development and testing
  - Increase estimation accuracy for project/release management
  - Increase experience sharing cross companies
- Participants
  - Company X initiator: Person A, Person B, Person C
  - Company X current contact person: Person D
  - NTNU: Reidar Conradi, Tor Stålhane, Jingyue Li, Jan M. W. Kristiansen (2009-2010)
  - EVISOFT: Tor Ulsund/ later Per Øyvind Markussen (EVISOFT project manager)

6

Original research plan and activities

ID	Activity name	Company Effort*	R&D effort*
A1	**Administration**	50+150	50+30
A2	**Improved Defect Management (done):** revised defect classification system and related processes to capture, analyze and follow up defects	150+570	50+120
A3	**Improved Release Management :** improved overall release processes for planning of configurations, testing (especially) and reuse	320+860	75+100
A4	**Improved SPI Management:** gap analysis (done), benchmarking, PMAs and use of wiki technology to manage and disseminate revised processes	110+570	50+100
A5	**Dissemination** – internally and externally	30+70	25+100
	SUM for 2007:	660	250
	SUM for 2008:	2220	450

* Person-hours in 2007 and 2008

* Person-hours in 2009 is not summarized yet

S1: Gap analysis (Jan. - May 2008)

- Gap analysis is an internal status survey with an important extra question added:

  How important is this for the future?

- This enables us to tell how far away we are – how difficult is it to get there.

8

S1: Choosing improvement goals

Improvement activities should in our case be chosen based on:

- Results from the gap analysis – what is considered important?
- Cost and benefits – is it worth it? SPI is a company investment like any other.
- The activity's ability to support the company's overall business goals – will this lead us in the right direction?

9

S1: Some gap analysis results

- Items ranked as important by *all* participants:
  - The ability to develop products/services that can be sold
  - The ability to identify market needs
  - An established process that can be used to understand the customer's expectations to the product
  - The ability to estimate needed effort
  - The ability to think long-term quality instead of short-term problem solutions

10

S1: Some gap analysis results (cont')

- Items ranked as important by Company X *developers only*:
  - The ability to make optimal code in relation to quality, maintainability and functionality
  - The ability to avoid risk-prone solutions
  - A process/standard for module testing
  - Failures/defects are reported in a formal way
  - Defect management is prioritized
- But no decision on a general, internal SPI initiative.

11

S2: Goals of BTS Improvement

- Gap analysis shows the following important items:
  - The ability to avoid risk-prone solutions (development)
  - Failures/defects are reported in a formal way (development)
  - Defects are prioritized (development)
- Motivation of the BTS improvement
  - *Provide timely, relevant, reliable, and easy-to-analyze defect information for evidence-based software quality assessment and software process improvement*
  - *"**4-2-4 rule**": 40% of Company X developers works with **testing**: room for improvements?*

12

S2: Foci of BTS improvement

- To answer the following Questions (Qs):
  - Q1: Effectiveness of testing methods (Which testing methods discover most defects before release?)
  - Q2: Testing should be improved (Which testing methods reproduce most defects reported by customers?)
  - Q3: Most time-consuming defects (Which defects are the most time consuming ones to be fixed?)
  - Q4: Possible early prevention (Which could be done to prevent the defects in early phases of a project?)
- Use IBM's Orthogonal Defect Classification (ODC) to extend and customize a defect classification scheme.

13

S2: Implemented BTS extensions as extra "ODC" attributes

- Q1: "Which testing methods discover most pre-release defects?" and Q2: "Which testing methods reproduce most post-release defects?"
  - Activity to discover defects, with values such as *performance testing*, *security regression testing*, and *URL top-list browsing testing* etc.
- Q3: "Which defects are the most time-consuming ones to be fixed?"
  - Effort to reproduce, locate and fix, with 4-5 ordinal values such as *easy-fix* and *time-consuming* – not 12:53:97 person-hours!
- Q4: "How to prevent (ripple) defects in early project phases?"
  - Defect type, with values as *algorithm*, *assignment*, *relationship* etc.
  - Root causes, with values as *requirement*, *design*, *code, …*

14

Status of using the ODC defect classification scheme

- ODC defect attributes introduced in May 2009.
- 80+ projects used ODC defect classification scheme, and classified over 2137+ defects.
- Core and Desktop projects were the main users; other projects used it less frequently.

15

S2: Some Analysis activities

1. Downloaded 836+ defect reports from Core and Desktop projects for analysis (by NTNU)
   - 166 classified as *time-consuming;* 670 as *quick-fix*
2. Read the log of *time-consuming* defects and classified these manually (by NTNU)
3. Performed email survey to collect feedback (by Person A at Company X): result??

16

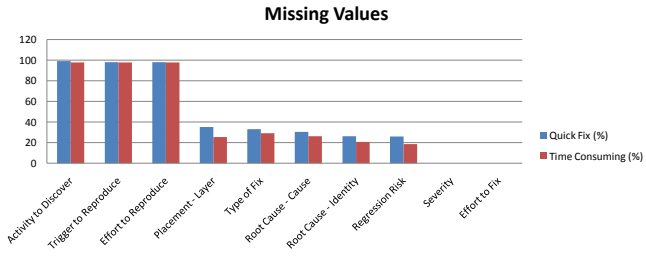
S2: Findings from analysis of filled-in defect attributes

- *Time-consuming* defects with top priority (P1) covered 28% of the Core project and 20% of the Desktop projects.
- 35% of *time-consuming* defects in the Core project were attributed to *design*, 10% of the *quick-fix* ones in the Core were about *design*.
- 28% of the defects were classified as *checking*, followed by 22% as *algorithm*, and 15% as *assignment* in these two projects.
- Thus:
  - It is necessary to reduce the number of time-consuming defects
  - It is necessary to review the design more formally
  - It is necessary to introduce code review, since *checking* and *assignment* defects can be easily removed by quick code reviews

17

S2: Observations from analyzing defect classification data

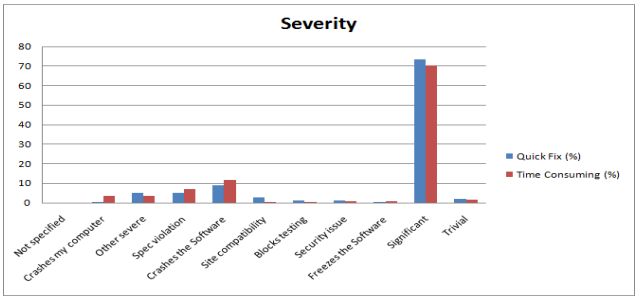
- *Missing values*: 20-40 % of every ODC attribute have their values missing (even if 91 % of such defect reports are “resolved”).



18

S2: Observations … (cont')

- Default values severely biased results
  - **Ex.** *Severity* have the automatic default value *Significant*, and 80% defects have such a severity



19

S2: Findings from reading debugging log of *time-consuming* defects

Reasons for the costly debugging	Number of cases		Suggestions
Hard to determine the location of the defect			ther analyze ts
Long clarification (discussion) of defect			ilitate king process in the core project
Implemented functionality was new or needed a heavy rewrite	12	23	Need to improve design of desktop project
The original fix introduces new defects / multiple fixes	9	6	Better regression testing in both projects
Reasons are not clear	4	9	
Others (documentation is incorrect or communication is bad)	2		

Four kinds of *time-consuming*:
-Time consuming to *locate*
-Time consuming to *decide*
-Time consuming to *fix*
- Time consuming from *ripple* effects

S2: Feedback from BTS users

- Defect classification needs to be more customized and updated according to Company X's context
- Training is needed
- Regular feedback channels are missing (like today's meeting)
- Change some attribute values from single- to set-values (multiple choice)

21

Recommendations: Improve the BTS further

- The system is on the right track, but small improvements still needed:
  - All relevant defect classification data should be filled-in, enabling better prevention of critical defects and less time-consuming debugging.
- Further update the ODC classification scheme based on business goals of the company, our analyses and users' feedback
- Improve the workflow of BTS:
  - I.e., let "right person" fill in "right data" at the "right time"

22

Reminder: Success or failure of SPI

Success is achieved by:

- Involving personnel in the whole process
- Commitment from management
- Start with simple, but highly visible improvements
- Steady course

We need support from management and give training to BTS users

Failure is achieved by:

- Uninterested management
- Frequent change of course
- Trying to solve all problems at once

23

Discussions

- Plan to improve the BTS further with defect classification scheme
- Further analysis on the time-consuming defects, especially those in the B2B projects
- Other current foci of Company X?

6/11/2010 Manager feedback meeting 24

APPENDIX D

MEETING MINUTES APRIL 29TH 2010

The meeting was held as a follow up and status meeting from the previous meeting at the March 23rd 2010. The minutes have the name of Company X replaced with "Company X". The minutes were written by Li, Conradi and representatives from Company X.

Minutes from meeting at April 29th 2010

1. **S1. Adjustment of defect attributes (ODC)**: to correct/improve the defect classification scheme, based on feedbacks from users.
    1. Revise the default value of some attributes of the BTS system, for example severity
    2. Add online tips of the "metadata" classification attributes to make those attribute values/terms easy to be understood by the users
    3. Further revise the classification attribute values to make them precise, easy to be understood, and simple, and validate them in some projects
2. **S2. Task flow in general:** to address missing values in the improved BTS system, by revising the task flow around the BTS to let "right person" fill in "right data" at the "right time".
    1. Find a mean to remind developers/testers fill in the "metadata" at certain state of the bug reporting
3. **S3. Cost-benefit model for defects:** to address the "time-consuming" defects in projects to speed up project deliveries and to ensure software quality. We need to analyze the development and debugging cost data of projects to find out the possible problems and corresponding solutions.
    1. Check more B2B projects to see if there are more "metadata" filled-in by some projects. Currently, we have only access to three projects (i.e.) with limited "metadata".
4. **S4. Defect management a bit more general:** to analyze defect data from more projects to find out problems of software quality and possible solutions.
    1. After revision of the defect classification schema of the BTS and collect more data, the data will be analyzed for defect management purposes.
    2. The main purpose of the software quality improvement is to find out the bugs in core engine and UI layers as early as possible, before they were delivered to the delivery projects.
    3. Currently, need to roughly decide the software quality improvement focuses, in order to plan the expected data analysis methods
5. **S5. Present EVISOFT work:** to advertise the results from using the new ODC and the improved BTS, so that **more developers and testers will start using** these improvements.
    1. After the revision of the defect classification schema and validations in a few projects, developers and testers of the whole company will be trained to use the schema properly.
6. **S6. A trial of a new bug clone detector:** we have recently developed a tool to find cloned bugs across products in the product line. Since we found many **duplicated bugs** across products in the product line, we would like to try this tool to see whether it can help to reduce duplicated bugs. The prototype of the tool is almost ready; we just need some projects to try out the tool and validate it.
    1. Company X going to check if there is a high level of code clone/code similarity in the code and the feasibility of trying the tool
7. **S7. Find bug duplication across projects:** When a new bug is reported, the developers and testers want to know whether the bug is a new one or is a duplicated one that has been fixed by other projects. Currently, the system provide only limited key word search or test case search to find duplicated bugs. The results are not satisfactory.
    1. NTNU will investigate if there is any method to address this issue

APPENDIX E

SCRIPT FOR DEFECT REPORT ANALYSIS

This appendix lists a sample script of the technique that was used to extract data from defect reports. The script was originally developed for the specialisation project during the fall of 2009 in Kristiansen [2009].

```
# Necessary imports
from sys import stdin
from BeautifulSoup import BeautifulSoup
import re
import os

# The path ot the file
path = 'path_to_file'

# The name of the attribute to be parsed
attribute = 'atribute_name'

discoverdict = {
    'category 1' : 0,
    'category 2' : 0,
    'category 3' : 0,
    'category 4' : 0,
    'Not specified' : 0,
}

attributes = [discoverdict]

# Parses the attribute and returns a dictionary with either
# the attribute incremented or not specified incremented.
def parseCustomDataAttribute(element, attribute, dictionary):
        parent = element.find(text=re.compile(attribute))
        if parent:
            value = parent.findNext('td')
            print value
            inc = False
```

```
            for key in dictionary.keys():
                if value.find(text=re.compile(key)):
                    dictionary[key] = dictionary[key] + 1
                    inc = True
                    break
            if not inc:
                print "I DIDN'T FIND THIS VALUE!!"

        else:
            dictionary['Not specified'] = dictionary['Not
                specified'] + 1
        return dictionary

extList = [".htm",".html"]

bugList = os.listdir(path)
i = 0
for filename in bugList:
    if os.path.isfile(path + filename) and
        os.path.splitext(filename)[1] in extList:
        i = i + 1
        print "Processing: " + filename

        f = open(path + filename, 'r')
        contents = f.read()

        doc = BeautifulSoup(contents)

        # ODC data
        customdata = doc.find('table', id='tab2')
        discoverdict = parseCustomDataAttribute(customdata,
            attribute, discoverdict)


# prints the results
for dictionary in attributes:
    for key in dictionary.keys():
        print key + "\t" + str(dictionary[key])
    print "\n\n"
```

Listing E.1: *A sample script of how data was extracted.*

To run the script, one would type:

```
python scriptname.py > output.txt
```

Listing E.2: *How to run the script.*