Managing Software Quality through Empirical Analysis of Fault Detection

Carina Andersson

LUND UNIVERSITY
Department of Communication Systems
Faculty of Engineering

ISSN 1101-3931
ISRN LUTEDX/TETS--1079--SE+216P

Printed in Sweden
E-kop
Lund 2006

This thesis is submitted to the Research Education Board of the Faculty of Engineering at Lund University, in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Engineering.

Contact Information:

Carina Andersson
Department of Communication Systems
Lund University
P.O. Box 118
SE-221 00 LUND
Sweden

Tel: +46 46 222 33 19
Fax: +46 46 14 58 23
E-mail: carina.andersson@telecom.lth.se
http://serg.telecom.lth.se/personnel/carinaa/

Abstract

Software often constitutes a large share of today's products. To deliver a product with the level of software quality expected by the customer, software development organizations need to manage quality achievement and assessment.

In practice, quality is often referred to as the lack of bugs. The research in this thesis is focused on how to improve software quality through effective and efficient detection of those software faults, and how to assess the level of software quality through analysis of the software faults. The research also further evaluates previous work, for example by replications, to continue building empirical knowledge in the field of software engineering.

In a literature survey of empirical studies on inspections and testing, it is shown that the choice of detection method depends on several factors. Examples of factors that affect effectiveness and efficiency of fault detection include the type of fault and the type of artifact in which the faults occur. An experiment is conducted to compare the performance of inspections and testing on faults originating from the design. The fault detection effectiveness is also examined in a software process simulation study, validated with empirical data, to illustrate the impact of different factors in the process.

A large case study was launched iteratively in an industrial context, investigating three development projects where the product quality is measured by analysis of the detected software faults. The analysis is guided with the purpose of generalizing findings obtained from other research studies. Fault distributions are examined, in terms of detection phase, location of faults, and fault density. In addition, a selection method for software reliability growth models is evaluated by application to fault data.

The contribution of this thesis straddles both research and practice. The conclusions of the thesis with its replicative approach are that to generalize any finding it is necessary to explore the applicability of the techniques investigated by gradually changing their parameters in additional studies.

Acknowledgements

First of all I would like to thank my supervisor Per Runeson, for supporting and guiding me during the work of this thesis. I would also like to thank my second supervisor Thomas Thelin, especially for his support during the rough times.

I would like to thank my co-authors in each paper and all other people who have contributed to the research performed in this thesis.

Thanks to present and former colleagues at the Department of Communication Systems, for making it an excellent environment to work in. A special thanks to the members of the *Software Engineering Research Group*, without you there would not have been any research performed.

Finally, thanks to my family, friends and Jonas, who all have listened and cared. Thanks for putting up with me during this time!

Carina Andersson
Lund, March 2006

Contents

Introduction

Software often constitutes a large share of today's products. For the development organizations who deliver software products, quality is a key characteristic, in its own or its cost implications. Managing software quality is necessary to deliver products, which function in a useful, safe and reliable way. In this thesis research on support for software quality management is presented. First, the research is focused on how to improve quality during the development process, and second, on how to assess the achieved level of quality. The research is addressed in terms of that the quality is improved by detection of software faults and assessed by analysis of detected software faults. The efficiency and effectiveness of different methods for detection of faults are investigated and presented together with suggestions on how to further evaluate the performance of the methods. The research is discussed in a context by examples of related work. Furthermore, analyses of software faults detected in an industrial environment are performed, to investigate the benefits gained by measuring the faults and associated attributes during the development process. Research in terms of quantitative analysis of software faults is further evaluated in replications by application to new data obtained from large software projects developed in the industrial environment.

The first part of this thesis is an introductory part that summarises the work. The introduction is organised as follows: in Section 1, the research area is described, and work related to the research in the thesis is

presented. Section 2 discusses the research methodology used in the thesis. The main contributions of this thesis are presented in Section 3, together with some general threats to validity of the research conducted.

The second part of this thesis contains the research papers that constitute the main contribution of the thesis:

PAPER I. What Do We Know about Defect Detection Methods?

Per Runeson, Anneliese Andrews, Carina Andersson, Tomas Berling, Thomas Thelin

IEEE Software, 23(3) May/June 2006.

PAPER II. An Experimental Evaluation of Inspection and Testing for Detection of Design Faults

Carina Andersson, Thomas Thelin, Per Runeson, Nina Dzamashvili

Proceedings of 2nd International Symposium on Empirical Software Engineering, pp. 174-184, 2003.

PAPER III. Adaptation of a Simulation Model Template for Testing to an Industrial Project

Tomas Berling, Carina Andersson, Martin Höst, Christian Nyberg

Proceedings of 2003 Software Process Simulation Modeling Workshop, 2003.

PAPER IV. A Spiral Process Model for Case Studies on Software Quality Monitoring – Method and Metrics

Carina Andersson, Per Runeson

To appear in *Software Process: Improvement and Practice,* 2006.

PAPER V. A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems

Carina Andersson, Per Runeson

Submitted, 2006.

PAPER VI. A Replicated Empirical Study of a Selection Method for Software Reliability Growth Models

Carina Andersson

Submitted, 2006.

Contributions to the papers. For the included papers were I am the first author, the main contributions are performed by me as the order indicates. For Paper I, my contribution is the fundamental survey of research studies, which later formed the basis for the discussions presented in the paper. For Paper III, the first author and I developed the simulation model together, whereas a majority of the discussions with the organization representatives were performed by the first author.

Related publications. In addition to the papers included in the thesis, the author has contributed to the following papers:

- Verification and Validation in Industry – A Qualitative Survey on the State of Practice
  Carina Andersson, Per Runeson, *Proceedings of 1st International Symposium on Empirical Software Engineering*, pp. 37-47, 2002.
- Test Processes in Software Product Evolution – A Qualitative Survey on the State of Practice
  Per Runeson, Carina Andersson, Martin Höst, *Journal of Software Maintenance and Evolution: Research and Practice,* 15(1): 41-59, 2003. This paper is an extended version of the paper above.
- Understanding Software Processes through System Dynamics Simulation: A Case Study
  Carina Andersson, Lena Karlsson, Josef Nedstam, Martin Höst, Bertil I Nilsson, *Proceedings of 9th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pp. 41-48, 2002.
- Evaluating the Impact of Software Process Simulations – A Case Study
  Carina Andersson, Per Runeson, Thomas Thelin, *Proceedings of 2004 Software Process Simulation Modeling Workshop*, 2004.
- How much Information is Needed for Usage-Based Reading?
  – A Series of Experiments
  Thomas Thelin, Per Runeson, Claes Wohlin, Thomas Olsson, Carina Andersson, *Proceedings of 1st International Symposium on Empirical Software Engineering*, pp. 127-138, 2002.
- Evaluation of Usage-Based Reading – Conclusions after Three Experiments
  Thomas Thelin, Per Runeson, Claes Wohlin, Thomas Olsson, Carina Andersson, *Empirical Software Engineering: An International Journal*, 9(1): 77-110, 2004. This paper is an extended version of the paper above.

- A Replicated Experiment of Usage-Based and Checklist-Based Reading
  Thomas Thelin, Carina Andersson, Per Runeson, Nina Dzamashvili-Fogelström, *Proceedings of 2004 International Symposium on Software Metrics*, pp. 246-256, 2004.
- A Case Study on Quality Monitoring in a Highly Iterative Software Development Process
  Carina Andersson, Per Runeson, *Proceedings of 5th Swedish Conference on Software Engineering Research and Practice*, pp. 1-10, 2005. This paper is a shorter version of Paper IV.

1. Research areas

To detect software faults, organizations in the software development domain perform their verification and validation in various ways. In order to investigating the state of practice, a qualitative survey of 11 Swedish organizations was conducted [2][42]. However, a general view of how the verification and validation is performed is not easy to give. Instead, the purpose of the survey was to increase the understanding of current test process practices in software industry. In the survey it is concluded that larger organizations emphasized the well documented verification and validation process as a key asset, while the process was less visible and there was a lack of quantitative management in the smaller organizations. These organizations relied more on experienced people among the employees than on standards and documentation. The fact that larger companies tend to have a better defined software development process, is in line with a survey investigating current development practices in New Zealand [14].

A well defined process, which follows standards and is documented, include well-planned, comprehensive verification and validation activities. These activities are parts of formulated test strategies with the aim to ensure faults are detected before the software system is released [46]. Ensuring that the testing strategies are in place is a prerequisite of quality when developing large complex systems. The test process, techniques and tools are significant contributors to effective and efficient testing and high quality software. However, the test methods and tools as such do not guarantee effective testing and high quality software [33].

The research in this thesis investigates how to manage software quality in terms of effectively and efficiently detecting software faults, and how to

assess the level of software quality, in terms of analysing the software faults. The focus has impelled two *research themes*:

- Effectively and efficiently detect software faults as means for improving software quality
- Analysis of the detection of software faults as means for assessing the software quality

1.1 Software quality

Quality is a multi-faceted concept that is difficult to measure since it is not reflected by one single metric [12]. Two definitions for quality are available in the *IEEE Standard Glossary of Software Engineering Terminology* [16]:

> *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.*

The IEEE standard continues with defining quality attributes as the characteristics that affect a product's quality. According to *International Standards Organisation* (ISO/IEC 9126), software quality has a range of six attributes, such as *functionality*, *reliability*, *usability*, *efficiency*, *portability*, and *maintainability*, and a list of subcharacteristics to each attribute [17].

Despite this rich definition, software quality is commonly recognized as lack of "bugs" in the developed software product [23]. If the software contains too many faults, the basic requirement of providing the desired behaviour is not met. This practical definition of software quality is usually expressed in terms of *fault rate* and *reliability*. Fault rate (faults per thousand lines of code) is closely related to the development and testing process but not necessarily a good measure of the quality perceived by the customer. The customer rather relates to the number of failures they observe, than the faults that cause them. The frequency of the failures and their impact are often more closely connected to the customer's perception of quality [44].

Discussing quality in terms of software problems or bugs requires a precise definition of the problem's cause and symptoms. The following

definitions distinguish between the human action (an error), its result (a software fault or defect) and the manifestation of the fault (a failure) and are used in the thesis [16]:

> *Error:*
> *A human action shown by a mistake, misconception, or misunderstanding.*
>
> *Fault:*
> *Occurs when a human error results in a mistake in some software product. (In the thesis the terms defect and fault are used interchangeably).*
>
> *Failure:*
> *The departure of a system from its required behaviour.*

During test and operation, the behaviour of the software system is observed. When undesired behaviour occurs, a failure is detected. Hence, a fault in the code may not always produce a failure. A program of any size must be run for several years for the latent faults in it to be found. The reliability of a software system is described in terms of absence of failures. However, the reliability and failure behaviour are obviously affected by the number of faults existing in the software being executed [34]. Still, systems containing many faults may be reliable, because the faults are not triggered to manifest themselves as failures under the execution conditions given. Software reliability is discussed in more depth in Section 1.3.2.

1.2 Detection of software faults

To detect the software faults, which have been generated during the development process, two different strategies may be applied, static and dynamic strategies [46]. Techniques guided by a static strategy do not require that the system is executed and may be applied at all stages of the development process. These techniques may be applied as formal reviews like inspections [10], or automatic analyses of the code of a system or associated documents. However, to ensure that a program is operationally correct also techniques guided by a dynamic strategy, which mean that the system is executed with test data, are necessary [15]. On the other hand,

testing is only possible when a prototype or an executable version of a program is available. Both inspections and testing are activities contributing to validation and verification.

Validation:
Assuring that the correct system is developed.

Verification:
Assuring that the system meets its specification.

The detection of software faults is addressed in the first research theme in this thesis. The research scope in this thesis is within empirical research on the verification and validation process, with a focus on methods for fault detection and their impact on software quality. In particular the performance of inspection and testing activities are investigated in terms of the effectiveness and efficiency of the fault detection.

The following sections introduce the research topic and describe related work in the area.

1.2.1 Evaluation of fault detection techniques

The key issue in much of the empirical software engineering research is the investigation of objects in terms of one or more of the three factors, quality, cost, and schedule. If several alternatives for conducting a certain activity are available, which is the best one? Regarding fault detection activities, an extensive amount of studies on methods are conducted over the years, both methods for static analysis, like inspections, and dynamic, like testing. Surveys, presenting the verification and validation strategies separately, can be found in [3], which describes the state-of-art of inspections, and in [21], which describes empirical studies on testing.

A survey of empirical work investigating inspections and testing, with the focus on studies including evaluation of *both* inspections and testing is presented in Paper I. The survey covers case studies and experiments investigating the two fault detection methods. In addition to the survey of the methods, a discussion on how to further evaluate the methods is included. The difficulty of evaluating and synthesizing the results of various studies on the topic is referred to by Miller et al. [30] in 1995. They point out that in particular experimentation in software engineering needs to follow a framework to avoid too much variability. A large amount of the experimentation carried out is not comparable. More experiments[1] are inquired, although not those which are carried out in

isolation. Instead, other experiments should be considered, where it is possible to compare the results, e.g. replications[2]. Only then the findings can be generalized. Miller continues the discussion ten years later [31], further emphasizing the importance of replications. He suggests the software engineering field to build families of experiments to be able to produce reliable and generalizable results. Dybå et al. take a more practical viewpoint [8]. They argue that for the practitioners in software engineering to make informed decisions on for example which fault detection method to use, replications of experiments have an important role. However, also other types of studies should be conducted and published, such as evaluations conducted in industry.

Two industrial case studies[3], which investigate the relationship between inspection and testing techniques, are included in the literature survey in Paper I. In addition to these, several other case studies, investigating inspections and testing, report on experiences from industry. These often have a comprehensive viewpoint, though they are not presenting any statistically significant results. One observed distinction from the majority of the surveyed experiments is that often when the presented case studies evaluate the role of inspection vs. testing in finding faults, the inspection of other artifacts than code is also discussed. In contrast to several of the other experiments, which most often inspect code, for one experiment the inspected artifact is a design document. This experiment is presented in Paper II.

The most addressed question in the case studies investigating inspections and testing is whether inspections are worth spending effort on, compared to other fault detection activities. Most literature present data supporting the claim that inspections are more effective and specifically cost-effective for detecting and removing faults than having the detection and removal of the same faults in the later phases [23][43]. The general opinion from several development environments says that the use of inspections improves the product quality and that inspections reduce the testing effort [1].

1. For explanation of experiments, see Section 2.2.3.
2. Replications are further described in Section 2.1.
3. For explanation of case study, see Section 2.2.2.

1.2.2 Software process simulations for evaluation of fault detection

In addition to the research studies investigating fault detection techniques presented in Section 1.2.1 and the survey in Paper I, a different approach may be used for evaluation of fault detection techniques: software process simulations based on empirical data.

The simulation approach is useful for creating an understanding of the mechanisms affecting the development process. An implemented computer model, calibrated with empirical data, is executed, simulating the real process behaviour. The empirical issues regarding software process simulations are several. Analysis of the software development process data is the basis for the direct input to the model or used for the model building. The model output data might be used as support for planning and management decisions. Also, the model structure is used in the context of evaluating the efficiency of a process.

Hence, to use a simulation model, the model should be customized to an organization, to ensure that the right metrics are used as input. The published models do often follow a generic waterfall life cycle, presenting either the whole development process, from requirements to system testing, or chosen parts of it, e.g. the unit testing phase. One benefit of simulation modelling is that the activities have to be clearly identified to get an accurate model. Furthermore, dependencies, as well as the feedback between various activities, have to be defined. An additional benefit with simulation modelling is enhanced learning through experimentation on a model instead of the more risky alternative to experiment on a real system. This may provide insights into e.g. processes before time and cost are invested in a change. Software process simulation may also be of benefit in training [38].

Several studies describing software process simulation have been published. However, they are rarely empirically validated with industrial data and simultaneously describing inspections and testing, although exceptions exist [28][29][40].

Development of process simulation models requires determining the accurate input parameters, which could be representative distributions for key project parameters, such as productivity, fault generation rates, task effort, etc. The development also requires quantification of relationships between the key parameters, which should be able to implement into a model. Obviously, software process simulation models are only simplistic models representing the real world, and the model results are highly dependable on the accuracy of the input. Nevertheless, the software

process simulation models may be valuable when evaluating different fault detection techniques and their impact on the quality of the developed software system. This is investigated in Paper III.

1.3 Analysis of software faults

The second research theme investigated in this thesis concerns the analysis of detected software faults. The motivation for analysing the faults are multifold: to gain confidence in the developed software, to predict the number of residual faults in the system, to evaluate how successful both the development and test processes are, etc.

1.3.1 Defect metrics

A survey of current test practices in Australia shows that the most popular software testing metric is fault count [36]. Moreover, often only a rough measure of overall software quality based on existing data is needed. Then quality in terms of a more simple measure, such as the number of detected faults, is sufficient for software practitioners [12]. Fault data are most convenient to track for measuring and showing progress in the development of a product of known quality [13].

Fault density, i.e. relating the number of faults to the size of the developed software system, is a standard measure of software quality [12], rather often applied in practice. However, fault density is derived from the process of detecting faults. As a consequence, the fault density might define more of the quality of the fault detection and reporting than of the quality of the software system itself. Nonetheless, a number of different studies have utilized fault density in metrics programmes, e.g. [13][19].

A number of attributes are connected to each fault. Fenton and Pfleeger describe eight mutually independent attributes, suitable for categorisation of a fault report [12]. Table 1 presents the attributes given by Fenton and Pfleeger, whereas organizations sometimes define their own attributes. The diagnosis of each fault is in many instances manual recording, and obviously not all attributes are included in all data collection. Depending on the goal of the analysis some attributes are more useful than others to evaluate.

Several case studies focusing on metrics from the test process have been published. These studies take a practical view and describe the measurements conducted in an industrial context. Typically, the case

studies focus on modelling the test process [47], prediction of the number of faults or other quantitative analyses [22], in order to evaluate the quality of the developed software system [7]. In Paper IV the collection of defect metrics from three large industrial software projects are presented.

Fenton and Ohlsson make a comprehensive analysis of faults from two releases of a large telecommunications system [11]. In addition to their own analysis, they discuss related empirical work investigating the quality and reliability of commercial software systems. Fenton and Ohlsson are adding to the growing empirical knowledge of fault distributions and fault densities. The hypotheses they tested are further evaluated in Paper V.

Table 1. Attributes of a fault report [12].

Attribute	Description
Location	Within-system identifier, such as module or document name
Timing	Phases of development during which the fault was created, detected, and corrected
Symptom	Type of error message reported, or activity which revealed fault
End result	Failure caused by the fault
Mechanism	How the fault was created, detected, corrected
Cause	Type of human error that led to the fault
Severity	Severity of resulting or potential failure
Cost	Time or effort to locate and correct

1.3.2 Software reliability

As mentioned in Section 1.1, reliability is one attribute of software quality. Software reliability is formally defined as [34]:

> *The probability of failure-free operation of a software program for a specified time in a specified environment.*

According to the definition of reliability, in addition to the aspect of failures, a second aspect of software reliability measurement is time. The reliability quantities are related either to the execution time for a software system (i.e. the CPU time), or the calendar time. The third aspect of the software reliability definition concerns the execution environment. The environment is described by the operational profile. Musa et al. [34]

describe the concept of operational profiles. An operational profile consists of the set of operations that a system is designed to perform and their probabilities of occurrence. Thereby, a quantitative characterization of how the system will be used is provided, and the software quality is measured from the customer perspective.

Reliability modelling and prediction has grown to a research field in itself. Based on past observations the models try to predict future reliability. *Software reliability growth models* are usually applied during system test, where cycles of test executions, observed failures, repair, and continued testing are repeated. By observing the changes in failure rate over time, the software development company has a support in using these to make decision about when to stop testing, and to deliver the product.

The software reliability growth models used in this thesis are of the type non-homogenous Poisson process (NHPP), assuming that the observed failures occur as a random process and he failure intensity is not constant. As faults are detected and removed from the software, it is expected that the observed number of failures per time unit will decrease [12].

The analytical research on software reliability growth models (SRGMs) is extensive. A wide variety of models are proposed, with individual assumptions for each model. In this thesis the practical application of the models on data obtained from industry is of more interest than the need for a new, specific model. Practical experiences of the use of reliability growth models in a variety of contexts are published, e.g. by Ehrlich et al. [9], Wood [52][53], Stringfellow and Andrews [48], and Jeske and Zhang [18]. Stringfellow and Andrews address the problem of selecting the SRGM which is most appropriate for a specific data set. Paper VI further investigates their proposed selection method.

2. Research methodology

Empirical research in the area of software engineering has evolved over the last decades [55][45]. Empirical studies in software engineering have become a key approach for researchers who want to understand, evaluate and model the techniques and methods being developed for software engineering. An empirical study is basically a systematic observation, which lets the researcher gain quantitative or qualitative evidence

concerning the object under study. Thus the research will allow for confirmation of theories and hypotheses based on observations rather than belief.

In order to investigate the research themes presented in Section 1, general empirical research methodology is discussed in this section, and the specific techniques and means used in this thesis are described.

2.1 Research through replications

Even though the field of empirical software engineering has evolved, the research has been criticized for still having low standards [49][54]. However, recent guidelines for evaluating situations, methods and techniques are proposed from several directions [8][25][51].

Attempts to summarize the empirical results are done, for example in the area of inspections [3], test techniques [21], and software engineering in general [55]. The summaries serve as starting points, showing a growing body of knowledge about various aspects of software engineering. However, even with facts about several aspects in the research field, it is difficult to know how to put the pieces together [39]. Different pieces of evidence are needed to support decisions taken in the software development process. Therefore, to gain greater confidence, it is argued that the decisions should be taken based on a diversity of arguments.

A diversity of arguments could be gained by conducting or participating in families of research studies, rather than relying on single studies [39]. For example, by replicating a study, more generalizable results might be achieved, rather than isolated and uncertain findings [27]. Also, as further discussed in Section 2.2, the results from a single study is difficult to generalize, whichever investigation type is chosen.

A replication could be designed to extend the scope of the results obtained in a previous study, by varying the conditions under which the researcher repeats the study [27]. The aim of a replication is to investigate whether the same result occurs despite differences in the studies. By analysing the results, it is seen whether or not a degree of generalization could be achieved. A replication may either be *close*, where the result is expected to recur, or *differentiated*, where it is unknown whether the result will recur or not. All known conditions for a close replication are kept much the same, whereas for a differentiated replication deliberate variations in fairly major aspects of the conditions are involved. A basic

reason for running a differentiated replication is to increase the external validity. The threat to validity may be countered by triangulation, which is achieved by using various methods [41]. Thus, more confidence is obtained that the results is due to the specific variables under study. A second reason for running differentiated replications is the extension of the scope of the result. Hence, the subsequent practical applicability is increased, whether it is extended to other organizations, other development environments or other time periods.

Replications are either *internal*, or *external* [31]. Repetitions undertaken by the same researchers are included in the category of internal replications. External replication is undertaken by other researchers. Eventually, the most differentiated replications, i.e. in most cases external replications, will provide the bigger payoffs in terms of wider generalizations [27].

Replication is a cornerstone of empirical knowledge [27] and research methodology. The replications show the range of conditions under which the result is so far known to hold. Thus, the replications show under which conditions the behaviour of the investigated phenomenon is becoming predictable and under which it may be applied to practical problems. To extend the knowledge base, replication can be applied to any type of study in the software engineering field. It is mostly known for experimental studies [4][31], although in this thesis two replicated case studies are presented.

2.2 Methodological approach

Independent on whether the goal of a research study is to replicate another study or not, the purpose of the research project should be defined, a research approach should be chosen and an appropriate study design decided upon. The chosen design will include the choice of data collection methods and analysis methods.

The following sections describe components useful for planning and running a research project:

- *Research strategy.* Two principally different methodological approaches for research are described.
- *Research purpose.* A classification of what the study is trying to achieve is presented.

- *Research methods.* The data collection and analysis methods used in the studies included in this thesis are described.
- *Sampling strategy.* It is described from where the data will be collected.

Two approaches are mentioned when discussing research strategy, *fixed* and *flexible* research designs [41]. Fixed design refers to doing a large amount of pre-specification about what to do, and how to do it, before getting into the main part of the research study. The approach means that the researcher needs to know exactly what to do, and collect all data before starting to analyse it. The fixed design often relies on quantitative data and statistical analysis, in contrast to flexible design. The flexible design often results in qualitative data, typically non-numerical, and much less pre-specification is used; the design evolves as the research proceeds, and the data collection and analysis are intertwined.

The purpose of the research can be classified into *exploratory*, *descriptive*, *explanatory*, and *emancipatory* [41]. If the research aims at seeking new insights and exploring what happens in situations not yet well understood, it is classified as exploratory. The purpose is to assess phenomena in a new light and generate ideas and hypotheses for future research. Exploratory research is almost exclusively of flexible design. To classify the research as descriptive, it requires extensive previous knowledge of the situation, to portray an accurate profile of events or situations. The descriptive research uses flexible and/or fixed design. Explanatory research aims to explain a situation or problem, and the patterns relating to the researched situation. It also aims to identify relationships between aspects of the phenomenon being researched, by using flexible and/or fixed design. Emancipatory research is used to create opportunities and the will to engage in social action. The emancipatory research uses almost exclusively flexible design. However, the purpose of a particular study may be influenced by more than one of the four classifications: exploratory, descriptive, explanatory and emancipatory, and the purpose may also change during the study.

Dependent on the purpose of the research, an appropriate investigation type is chosen. Three major alternatives are recognized, *surveys*, *case studies*, and *experiments* [24][51].

2.2.1 Surveys

A survey is often referred to as a fixed design research strategy, though it may also be of flexible design [41]. The central features of surveys are the collection of data from a relatively large number of individuals, and the selection of representative samples of individuals. Surveys are very common to areas like social science, for example for analysing voting intentions. Many of the variables that influence the studied field are not controlled in a survey, as in other investigation methods. The primary means of gathering data in a survey are interviews and questionnaires, although data may also be collected from documentation. The results are analysed to be generalized to the sampled population. Hence, the sampling, i.e. the selection from the investigated population, has to be conducted with consideration. Due to the sampling, the results from an investigation performed in one organization are difficult to generalize to other organizations.

2.2.2 Case studies

A case study is of a flexible design research strategy, focused on the situation, individual, group, project or organization that the researcher is interested in. A case study is defined as using several methods for data collection, where qualitative data most invariably are collected, though; also quantitative data may be included [41]. A case study is an observational study, and the researcher does not have the same level of control as in an experiment, i.e. there are confounding factors, which are not entirely known or cannot be controlled, that may affect the result. However, a case study is performed in a real context and not in a laboratory, as experiments often are. On the other hand, a case study might be harder to interpret. A case study can monitor the effects in a typical situation under study, though they cannot be generalized to every situation.

Having a strong focus on the investigated situation may require running a case study during a long time period. Then the flexible design approach will be even more evident, involving that the focus of the study and the research questions will evolve during the investigation. Hence, the case study process has to have an iterative character, to stepwise adjust the goal and scope to the findings retrieved through the data collection and analysis.

2.2.3 Experiments

The experimental strategy is of the fixed design type. An experiment is an extremely focused study, since only a few variables can be handled. The purpose is to manipulate one or more variables and controlling the others at fixed levels. An experiment is usually conducted in a laboratory environment, with subjects assigned to different treatments at random. One of the treatments, the *control* treatment, is often the status quo, and the use of a new method or tool is compared with the control treatment. The results from a formal experiment may be easier to generalize than the results from surveys and case studies. However, the context of the experiment is of high importance, since experiments do not generalize outside the controlled experimental conditions.

2.2.4 Sampling

Sampling was mentioned as an important criterion in Section 2.2.1, describing sampling in surveys. However, the sampling is also important in the other investigation types.

Sampling could concern the population of people of interest, e.g. software testers in a development organization, but also include populations of situations, e.g. a series of development projects, events or times [41]. It is seldom feasible to survey, or investigate, the whole of a population, and then the sampling is a means to reduce the population and get a selection thereof. Hence, a sampling of the population is more feasible, to get a manageable size on the investigated population.

The sampling strategy may be based on *probability samples* and *non-probability samples*. For the first type, the statistical probability that a person will be included in the study is possible to specify, and is referred to as representative sampling. For non-probability sampling statistical interference is not possible to obtain.

The size of the sample affects the possibilities to generalize the results. The sample size should depend on the population's homogeneity. If the population is heterogeneous, a larger sample is required to be able to generalize the findings than if the population is more homogenous.

2.3 Data collection and analysis

Depending on the chosen investigation type for the empirical study, feasible methods for data collection and analysis are decided upon. Without proper analysis and interpretation of the collected material, the essence of the data will not be revealed nor be possible to communicate. However, there might not be a clear point in time when the data collection ends and analysis begins. In a flexible design, usually the data collection and analysis are overlapping, which also may result in a higher quality of both the collected material as well as the analysis. The overlapping may result in new insights and alternative explanations when not focusing on confirming predefined solutions [37]. On the contrary, in a fixed design study, the analysis is performed after all data are gathered.

The procedure for data collection can be chosen among a variety of methods. The following focuses on the instruments that have been most frequently used in the studies presented in this thesis.

2.3.1 Observations

Observations for data collection are typically used in exploratory research, to observe what is going on in a certain situation, and to watch the actions and behaviour of people. The observations are then described, analysed and interpreted. Much research in social science involves direct observations of humans, but also, for example, experiments in software engineering represent a kind of controlled observation.

The observations used in the large case study presented in the Papers IV, V and VI in this thesis is mainly of supportive type [41]. The observations are used to collect data that complement the data obtained from other sources. In Paper III, observations are also used, but more of a participatory type [5][41], where the observer participates in the group or situation under study. In this study one of the researchers participated in the daily work in the organization.

Most important with observational methods is the awareness of the risk of bias, i.e. that the observer loses the objectiveness. When observing an organization, which the researchers themselves belong to, they have some knowledge of how certain situations are handled and thereby they may unintentionally overlook some aspects. On the other hand, the researchers often have a good understanding of the existing procedures in the observed organization.

2.3.2 Interviews

By interviewing the researcher asks questions and receives answers from the interviewees. One advantage with data collection through interviews is the flexibility. The researcher has the possibility to follow up on ideas, interpret feelings and facial expressions and intonations of the interview subject that written answers do not reveal. The answers given in a questionnaire must be interpreted on their own, while in an interview, attendant questions can be given and the answers thereby catch more subtle information. On the other hand, interviews are rather time consuming. There are several necessary activities that should be conducted; the preparation, the execution, and the processing of the data. It is also a subjective technique, with risk of biases, both from the interviewer and interviewees' point of view.

The classification of different interview types can be shown on a scale of structure, from one extreme, the *fully-structured* interview, over the *semi-structured* interview, to the other extreme, the *unstructured* interview [41]. The more standardized the interview is, the easier is the processing of the data. The fully-structured interview resembles a questionnaire or a checklist, though may also include open-response questions. The semi-structured interview has predetermined questions, but the interviewer may change the order, as well as the wording of the question, and explanations may be given. The unstructured interview has often a topic, which the interviewer poses open questions about.

In this thesis, the performed interviews are mostly of unstructured type (Papers III, IV and V).

2.3.3 Content analysis

A third method for data collection is review of written documents [41]. The content analysis in a qualitative or quantitative inquiry examines relevant records and documents, with the focus to gather information and generate findings. The analysis of what is in the documents differs from observations and interviews in that it is indirect. Instead of directly observing or interviewing, the content analysis is based on existing documents. In this case the observer does not affect the documents. In the study presented in Papers IV, V and VI, and in the simulation study presented in Paper III, the primary data collection have been through content analysis; data collected from existing documents and databases.

Content analysis also includes analysis of the content of interviews and observations, and then the data are collected directly for the purpose of the research. Content analysis has been used for this purpose in the literature survey in Paper I and analysis of the experiment data in Paper II.

2.3.4 Simulations

Simulations can be used both as a data collection method [41] and as an analysis tool, when experiments and real-world trials are too expensive and difficult to conduct. In this thesis, simulations are mainly used as support for the analysis. A model of the essential structure of the situation of interest is designed and computer simulated, though it is still only a model of the real world. Simulation of software development processes can be used for extending the understanding of the interdependencies between different activities in the process, or be used as means for an organization to evaluate process changes, etc.

Simulations are classified in two different types, *discrete-event* and *continuous* [26]. In discrete-event simulation, the state of the system is changed only when certain events occur. In continuous simulations, also referred to as system dynamics, the state changes continuously over time, and the simulation model is defined by differential equations. Software process simulation and modelling using the system dynamics paradigm have been applied in the study presented in Paper III, for increasing the understanding of the investigated development and test process.

2.4 Validity

Although a research study has been conducted with reliable, well-defined methods and techniques, the results and conclusions should be evaluated and questioned. The validity should always be addressed. The researcher is obliged to describe the used methodology and data collection procedures to a sufficient level of detail, and also how the researcher has reached the presented conclusions from the data. This to ensure that the quality of the conclusions can be assessed, and also to facilitate replication. The validity of a research study can be evaluated from four perspectives [50]:

Conclusion validity is concerned with the relationship between the cause and the effect in the study, and answers e.g. the question whether

there is any relationship between the treatment and the outcome in an experiment.

Internal validity is concerned with what really have had impact on the resulting outcome of a study. If a relationship between the treatment and the outcome is observed, the internal validity reflects if a particular treatment really has caused a certain outcome. One example of threats to the internal validity is the history effect, whether something has changed in the participants' environment, a change that is not part of the study.

Construct validity reflects whether the researcher measures what is decided on. A threat to the construct validity concerns if it is possible to generalize the results of an experiment to the underlying theory or hypothesis.

External validity concerns the generalizability of the study. Results obtained in a specific setting, or with a specific group of subjects, may not be representative for other settings. For example, results from a study in a laboratory setting can be difficult to generalize to other conditions, which are not close to the laboratories.

For these four validity types there are always several possible threats. Ideally, these threats are reduced as much as possible.

Threats to validity of the results in this thesis are discussed separately in each included paper, and in addition to that in conjunction with the main contributions in Section 3.5.

3. Contributions

This section presents the research that constitutes the main contributions of this thesis. In addition, the main threats to validity that have been addressed during the studies are discussed, as well as potential future research directions.

3.1 Research overview

Two research themes are addressed in this thesis. First, how to manage software quality, through effective and efficient detection of software faults. The goal is to gain knowledge on techniques used for software verification and validation. Evaluation of different methods are conducted, both in terms of different techniques' efficiency and effectiveness in fault detection and in terms of process factors' impact on

the process activities. As presented in Table 2, the first research theme is addressed by a variety of research methods. The methodological approaches used are a survey of available literature, an experiment evaluating two techniques, and a case study with simulation as a tool for analysis. The research contexts are both academic and industrial. The presented studies cover fault detection in a series of artifacts, ranging from requirements and design to code. The investigated parameters are varied to ensure that different viewpoints are reflected in the contributions.

The second research theme concerns how to assess software quality. The goal is to gain knowledge on how software quality could be measured. Software quality may be separated into a variety of attributes [17]. In the thesis, software quality from the perspective of detected software faults is considered. Hence, the quality is addressed by analysis of the detected software faults. The research is conducted in an industrial context, see Table 2, as a large case study with a number of cycles, each with individual goals. The projects included in the case study produced software systems, which are embedded products and relate to each other in a product-line context. The development organization belong to the telecommunication domain. The research methodology of a case study is specifically addressed and one of the contributions is a model for how a

Table 2. Research studies' purpose, method and context.

Paper	Research theme	Primary study purpose	Evaluation of	Research method	Context
I	Improve quality/ Detect faults	Descriptive	Techniques	Literature survey	Academia/ (industrial)
II		Descriptive	Techniques	Experiment	Academia
III		Explanatory	Factors' impact	Simulation	Industrial
IV	Assess quality/ Analyse faults	Exploratory/ Confirmatory	Relations: faults/time/ test activity	Case study	Industrial
V		Replication/ Descriptive	Relations: faults/location/size	Case study	Industrial
VI		Replication/ Prediction	Relations: faults/time SRGMs	Case study	Industrial

large case study like this may be performed. Furthermore, two of the case study cycles have been guided by the purpose to generalize findings obtained from other research studies in the software engineering field [11][48]. By performing replication studies, more reliable evidence on the investigated software engineering topic may be gained, since a single research study is often said to not being sufficient for producing reliable results [41]. In the thesis, research is performed to accumulate more knowledge on the studied topics, even though one single replication neither might be sufficient to make generalizations. Instead, the replications contribute to the understanding of the observed phenomena. By more studies on the investigated topics, generalisations could gradually be made.

3.2 Fault detection

The first research theme discusses how to achieve an adequate level of quality of the developed software products. The discussion is focused on having efficient and effective methods for fault detection, and how to choose which methods to use. The contribution to this research theme is presented in Papers I-III.

3.2.1 What do we know about defect detection methods?

Paper I surveys published empirical studies, which compare testing and inspection techniques. Ten experiments and two case studies are analysed, in an attempt to characterize the fault detection methods in terms of efficiency and effectiveness, and to give guidelines on which method to choose.

Not surprisingly, a clear answer on which method to choose could not be given. The results, and thereby the choice of method, depend on factors, such as the artifacts, the type of defects that the artifacts contain, who is doing the detection, and how it is done. In the paper, the results are separated with respect to type of defects:

- Requirements defects
- Design defects
- Code defects

Regarding requirements defects, the choice is quite obvious; spending effort to establish a good set of requirements early on is more efficient, than developing the system based on incorrect requirements and then rework it. Regarding design defects, one single experiment analyses defect detection in design inspections and function test. The inspections were significantly more efficient and effective than the testing technique. For more details on this experiment, see Paper II. The result was confirmed by the two case studies. Regarding code defects, a rather large number of experiments have investigated the issue, however, there is no clear answer whether code inspection or function or structural testing is preferable for detecting code defects. The results tend to favour testing in terms of effectiveness and efficiency, although secondary issues such as information spreading effects of inspections, value of unit test automation and costs of a test suite or test driven design, are not taken into account.

Even though no clear answers are given in the survey, the collection of studies presented may contribute to the understanding of the phenomena. Listed factors show that there are many variations to take into account when deciding on which method to choose. The factors should also be considered when deciding upon conducting new studies, either experiments or case studies. Evidence-based software engineering (EBSE) [8] is suggested as an approach to support these decisions. The non-homogenous findings of the survey indicate that both further research in the area and industrial evaluation have to be conducted, where the direction could be guided by EBSE. Specifically if industrial evaluation is fed back to the research community, the knowledge and the practical utility of the methods will increase and more informed decision might be made. But also further work inside the research community in terms of experiments, with varying factors, will provide more solid empirical evidence.

3.2.2 An experimental evaluation of inspection and testing for detection of design faults

As discussed in Section 1.2, a variety of both static and dynamic strategies is means for achieving a high level of quality in software product. The study presented in Paper II is included in the literature survey in Paper I. The focus of Paper II is on inspections and testing as detection activities of design faults, where representatives of techniques for these activities were investigated and compared. The techniques were evaluated in terms

of the fault detection capabilities, in a controlled experiment. As mentioned in Section 3.2.1, related work, i.e. experiments comparing the two detection methods, has focused on fault detection in code artifacts. Meanwhile, the work in this study emphasizes the importance of also investigating and comparing the activities on a higher abstraction level, in this case the detection of design faults.

The general results from this study show that efficiency and effectiveness are higher for the inspection technique and that the testing technique requires more time for learning. Although rework was not taken into account, i.e. the study included only fault detection and isolation of the faults, inspections were more efficient and effective. If rework is considered, the difference in efficiency would probably be even higher, since the correction of the faults detected by inspection is conducted earlier in the development process than the corresponding task after detection by testing. However, a combination of inspection and testing activities should be emphasized if the techniques detect different faults.

3.2.3 Adaptation of a simulation model template for testing to an industrial project

In addition to the investigation of specific inspection and testing techniques as fault detection methods above, the importance of having efficient and effective fault detection for improving the quality software products is investigated from the more general perspective of software development processes.

In this thesis the use of simulation as a tool for increasing the understanding of the software development process is investigated. Specifically, the simulation paradigm of system dynamics has been evaluated with respect to its usefulness as a tool for visualizing different factors' impact on the test process activities. The contribution of Paper III is the analysis of software process simulation as a tool for visualizing the fault detection in the various test phases, and the impact of the fault detection originating from both qualitative and quantitative factors in the various test phases.

Paper III describes how a template model is created in order to increase the knowledge of the code development and test processes for an industrial organization. The template model is created from an existing system dynamics model for the unit testing phase. The study shows how

the template model is adapted and extended to fit an organization. The simulation model is applied to investigate the relationship between fault prevention in the development phase and fault detection in the various test phases. Data from a large contract-driven project were used in a case study to calibrate the adapted and thereafter extended model, which included code development and four test phases. Programmers and testers were involved in the design of the model.

The results show that it is possible to use the introduced template model and to adapt and extend it to a specific organization. It is also concluded that it is important to involve project members who contribute to the model building. The process understanding among the participating project members is increased due to their involvement. Knowledge is gained specifically on which factors do have an impact on the fault detection.

3.3 Fault analysis

The second research theme in the thesis focuses on aspects of assessing the software quality of developed products, specifically in terms of the number of detected faults, which are analysed in various respects.

Papers IV-VI describe one organization, where data are collected from three software development projects, in one large case study. The case study was run during approximately one year, with several cycles with different data analyses in each. Paper IV describes the case study on quality monitoring as such in the investigated organization. The paper presents the chosen case study methodology and in addition, discusses the data analyses conducted in the first cycles. Paper V and Paper VI describe the results of two individual cycles in the case study process.

3.3.1 A spiral process model for case studies on software quality monitoring – method and metrics

The contribution of Paper IV is twofold. A spiral process model for an iterative case study is proposed. The case study process was developed to be able to properly manage the investigation of a highly iterative software development environment and the interfaces between researchers and representatives from the investigated organization.

By using the proposed case study process, goals and scope are adjusted iteratively as more information becomes available. By following the

proposed process, the activities of setting goal and scope, collecting and analysing data, the presentation and the interpretation of the analysis, are conducted in each cycle of the process. The representatives from the investigated organization have their own areas of responsibility in each process cycle: to continue a deeper interpretation of the data and based on the results take actions on potential software process improvements.

In addition to the proposed case study process, Paper IV presents the process when applied to an empirical study. The case study cycles range from exploratory type, while investigating the available information in the organization. After that, a confirmatory approach is taken, where the observed trends of the data from the previous cycles are statistically tested. Third, an explanatory approach is taken, where the observed behaviour of the collected data is analysed in more detail. Finally, prediction models are applied in additional explanatory cycles, to explain and predict future behavior of the investigated projects.

Mainly quantitative fault data were collected and analysed, but also qualitative data are used in the study. Specifically in the explanatory case study cycles, qualitative data were collected to explain the observed fault distributions, and to characterize the investigated projects and their feature groups.

The data have been analysed from different perspectives, however, the goal of the analysis in Paper IV is that it should be guided by simplicity. The collected data should be easily available, preferably already existing in the organization, not to burden the staff with additional data collection. Nevertheless, the results show that also this approach gives valuable input. For example, the observed fault distributions are used as rules of thumb for fault prediction. Applied at one of the milestones in the projects, approximately when system test starts, a prediction of the number of faults to expect at the delivery date of the software system is achieved.

The result from a case study like this has another field of application. The observed fault distributions, also including those which are presented in Paper V, could be used as a baseline, e.g. to evaluate newly applied test strategies and test methods introduced in a new project.

3.3.2 A replicated quantitative analysis of fault distributions in complex software systems

In Paper V the data from the investigated projects from the organization in Paper IV are further analysed. The analysis of the fault distributions uses a quantitative approach, and shows several aspects for how to assess the quality. The analysis is performed as a replication of previous work [11], with the goal to gradually generalize the findings.

A number of hypotheses concerning fault-proneness, location of the faults, and module size are tested. The hypotheses are tested to provide information on fault distributions from this specific development environment. The replicating approach of testing the same hypotheses as has been done before, gives surplus value by application to data from a different software system from a different development environment. The results of the hypotheses on fault distributions show that one study is not enough. Generally, the replicated study differs a lot from the original study with respect to size of the developed system, type of application, and number of faults. Hence, there is substantial variation between the studies, and also between the studied projects in the replication study. Still, the study adds to confirmation of quite general statements.

The quality of the investigated software systems is in the study assessed in terms of quantitative analyses of the systems' fault distributions. The fault distributions are analysed from four aspects:

1. Which modules the faults are located in, i.e. whether the faults are distributed according to a Pareto principle [20] is analysed. It is shown that a small number of modules contain most of the detected faults. Additionally, this could not be explained by the size of the modules.

2. The fault distributions are tested to show whether modules with high fault incidence in early test phases will have high fault incidence also in later test phases. The study supported the hypotheses that a high fault incidence in a module in function test implies the same in system test, and a high pre-release fault incidence imply high post-release fault incidence.

3. The module size effect on fault-proneness is analysed. In this aspect varying results were obtained. The relation between number of faults and lines of code were not consistent across the studied projects, and clearly the hypotheses could not catch the essence of

this issue. Other factors not taken into account in the study have had impact on the findings.

4. The fault densities across different releases and test phases are analysed. The hypotheses regarding constant fault densities were confirmed. It is shown that across projects and test phases the fault densities are similar. Some dissimilarity between the different types of projects is observed. However, the similarities between the projects are substantial, when analysing the percentage distributions of faults across test phases.

By analysing the received fault distributions from the investigated projects, comparisons between releases of systems may be conducted. The fault profiles do change between the different systems, but most often the same trends are observed. By observing the fault distributions in future projects, discrepancies may be sought for. Either because of deliberately conducted changes in the test strategy, or other changes not that apparent.

3.3.3 A replicated empirical study of a selection method for software reliability growth models

Paper VI also presents a research study, which uses a replicating approach. Similar to Paper V, the goal is to further evaluate a previously tested method in a different context, and thereby gain more confidence in the results.

As discussed in Section 1.3.2, software reliability is an attribute of software quality. In Paper VI, software reliability growth models are applied to assess the reliability and hence, evaluate the quality of the developed software systems in the investigated projects. The study evaluates and applies a selection method for software reliability growth models, used for assessing the quality of the developed software system and predicting the future outcome. Originally, the selection method was developed for making stop test decisions [48]. During system test the software system is evaluated with the selection method, and the predictions of total number of software faults, given by the software reliability growth models, are compared to a stop test decision criteria.

In the context described in Paper VI, the selection method is applied afterwards, and not in real-time. Thus, the method is not used in reality for the stop test decisions, but the method as such is evaluated. Variations compared to the original study imply gradual generalization of the

findings. In this replication study the method is mainly unchanged and the main difference to the original study is the application system. The method is applied to the investigated projects, which originate from a different development environment compared to the original study. The project duration differs; in the replication study the number of test weeks are approximately twice the number of the original study. In addition, the total number of faults differs from the original study and is approximately 10-fold larger in the replication study. The extension of the original study to a completely different context contributes to investigating the generality of the original study.

The replicated evaluation of the selection method showed that, with some changed values of the parameters in the procedure, the selection method worked well. That is, the method was applicable in another environment than the original one.

Assessing the quality of a software system often results in fault or failure count. From this perspective, predictions of the number of remaining faults in the investigated system are of high interest. In Paper VI, the prediction ability of the software reliability growth models is investigated. The results show that for the investigated systems, the predictions, when closing up to release dates, are rather good. However, it is questioned whether the given estimates are reliable when asked for, which usually is several weeks before release. Compared to the predictions of the number of faults to expect presented in Paper IV, reliable predictions from the software reliability growth models are received later in the process. However, the two presented prediction approaches are not supposed to be compared as two methods giving the same prediction result. They should be seen as two individual methods with different purpose, which may both be applied, but at different occasions during the software development and testing process.

3.4 Summary and further work

Obviously, not all research studies can be unique investigations. The research in this thesis has focused on further evaluation of previous work, to continue to build empirical knowledge in software engineering. The research themes are focused on software quality management through empirical evaluation of fault detection and analysis. For both of the addressed research themes it is shown that in order to generalize any findings it is necessary to continue the research by gradually changing the

investigated parameters in additional studies. By that we may base our decisions in the software development process on facts that are explained and understandable.

For the first research theme, regarding software fault detection, various research studies point out that the effectiveness and efficiency of fault detection methods are dependent on the artifact where the faults are searched for. Continued research in fault detection methods will add to the already available knowledge even further. Software process simulations have been shown to increase the understanding of different factors' effect in the development process. In addition, as discussed in Paper I, further research in the area of effectiveness and efficiency of fault detection methods may be guided by evidence based software engineering [8]. Optimally, more aspects may be investigated, in particular those specific to an industrial context. However, also research conducted in academic environments, such as controlled experiments will give valuable insights.

For the second research theme addressed in this thesis, analysis of software faults, several more aspects may be investigated. It is shown that the results are dependent on the type of software system and its development environment. Hence, to create more general knowledge, further research by varying these parameters are necessary. The variation may concern the software system and the development context, but also other principles of the investigation. For example, the analysed aspects of the fault distributions presented in Paper V may possibly be extended with other approaches for assessing the software systems. Newly formulated hypotheses may be tested, either on a similar software system or e.g. on a system developed for a completely different use. Regarding the selection method of software reliability growth models presented in Paper VI, the method may be applied on different software systems, but also evaluated in terms of including a different set of SRGMs, and by varying the evaluation criteria.

3.5 Validity threats

The contributions discussed in the previous sections rely on the conclusions drawn from the results of the studies, which are reported in the included papers. The threats to validity are discussed separately in each paper, together with applied strategies to reduce the threats. In addition, the main validity threats to the contributions are discussed below.

To generalize any finding beyond the setting studied, external validity is of importance. In for example a case study, the goal is possibly just to explain or understand the investigated setting. However, with the purpose to generalize beyond the specific setting, a selection of a representative sample is necessary to obtain statistical generalization. Some type of sampling has occurred in all studies included in this thesis. The sampling has concerned the type where you select people from a population, but also projects in a development organization, and faults from a database with software problems. The balance between the need of being selective and the need to collect the required data to answer the stated research questions has been attended to. Variation has been sought for, even though convenience sampling has occurred when no alternative has been given. Convenience sampling is not the most satisfactory way of sampling, but the research is instead intended to present views of the chosen settings. By further research, for example by replications, more views could be given to continue to build the empirical knowledge in software engineering and eventually generalizations of the results may be expressed. Replications are advocated as research approach in this thesis, and applied in practice in two of studies. These examples are conducted as external replications, i.e. conducted in a different environment and by different researchers compared to the original studies. As replications, threats to external validity is decreased.

Accordingly, a single case study as research methodological approach, states that the results are not generalizable to every other setting. The conditions specific to for example the investigated organization influence the outcome. The results of the case study presented in Papers IV-VI are obtained from an environment represented by one single organization, which develops software products in one domain. The type of products are the same, and also approximately of the same size. On the other hand, the internal generalizability, i.e. within the organization, in the presented studies is increased by analysis of data from three separate development projects. A majority of the presented studies has been conducted as case studies, and as such approach, not every parameter could be controlled, which decreases the internal validity. Without control of every factor a causal relationship cannot be established for sure. The proposed case study process attempts to reduce threats to validity by continuous feedback to representatives from the investigated organization. However, there are potential threats for bias, both from researchers and industry participants. These threats were countered by triangulation [41], i.e.

having multiple sources for the data, and member checking when material was fed back to the organization.

4. References

[1] Ackerman, A. F., Buchwald, L. S. and Lewski, F. H., "Software Inspections: An Effective Verification Process", *IEEE Software*, 6(3): 31-36, 1989.

[2] Andersson, C. and Runeson, P., "Verification and Validation in Industry – A Qualitative Survey on the State of Practice", *Proceeding of the 1st International Symposium on Empirical Software Engineering*, pp. 37-47, 2002.

[3] Aurum, A., Petersson, H. and Wohlin, C., "State-of-the-Art: Software Inspections after 25 Years", *Software Testing, Verification and Reliability*, 12(3): 133-154, 2002.

[4] Basili, V. R., Shull, F. and Lanubile, F., "Building Knowledge through Families of Experiments", *IEEE Transactions on Software Engineering*, 25(4): 456-473, 1999.

[5] Bell, J., *Doing Your Research Projects* (3rd ed.), Open U.P., 1999.

[6] Boehm, B. W., *Software Engineering Economics*, Prentice-Hall, 1981.

[7] Daskalantonakis, M. K., "A Practical View of Software Measurement and Implementation Experiences Within Motorola", *IEEE Transactions on Software Engineering*, 18(11): 998-1010, 1992.

[8] Dybå, T., Kitchenham, B. A. and Jørgensen, M., "Evidence-Based Software Engineering for Practitioners", *IEEE Software*, 22(1): 58-65, 2005.

[9] Ehrlich, W. K., Stampfel, J. P. and Wu, J. R., "Application of Software Reliability Modeling to Product Quality and Test Process", *Proceedings of the 12th International Conference on Software Engineering*, pp. 108-116, 1990.

[10] Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development", *IBM Systems Journal*, 15(3): 182-211, 1976.

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

[12] Fenton, N. E. and Pfleeger, S. L., *Software Metrics: A Rigorous and Practical Approach*, Thomson Computer Press, 1996.

[13] Grady, R. B., *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, 1992.

[14] Groves, L., Nickson, R., Reeve, G., Reeves, S. and Utting, M., "A Survey of Software Development Practices in the New Zealand Software Industry", *Proceedings of 2000 Australian Software Engineering Conference*, pp. 189-201, 2000.

[15] Hetzel, B., *The Complete Guide to Software Testing*, John Wiley & Sons, 1988.

[16] Institute of Electrical and Electronics Engineers, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990.

[17] International Standards Organisation, *Information Technology – Software Product Evaluation – Quality Characteristics and Guide Lines for their Use*, ISO/IEC FDIS 9126-1, 2000.

[18] Jeske, D. R. and Zhang, X., "Some Successful Approaches to Software Reliability Modeling in Industry, *The journal of Systems and Software*, 74(1): 85-99, 2005.

[19] Jones, T. C., *Estimating Software Costs*, McGraw Hill, 1998.
[20] Juran, J. M. and Gryna Jr. F. M., *Quality Control Handbook* (4th ed.), McGraw Hill, 1988.
[21] Juristo N., Moreno A. M. and Vegas S., "A Survey on Testing Technique Empirical Studies: How Limited is Our Knowledge", *Proceedings of the 1st International Symposium on Empirical Software Engineering*, pp. 161-172, 2002.
[22] Kaâniche, M. and Kanoun, K., "Reliability of a Commercial Telecommunication System", *Proceedings of International Symposium on Software Reliability Engineering*, pp. 207-212, 1996.
[23] Kan, S. H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995.
[24] Kitchenham, B. A., Pickard, L. M. and Pfleeger, S. L., "Case Studies for Method and Tool Evaluation", *IEEE Software*, 12(4): 52-62, 1995.
[25] Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K. and Rosenberg, J., "Preliminary Guidelines for Empirical Research in Software Engineering", *IEEE Transactions on Software Engineering*, 28(8): 721-734, 2002.
[26] Law, A. M. and Kelton, W. D., *Simulation Modeling and Analysis* (3rd ed.), McGraw-Hill, 2000.
[27] Lindsay, R. M. and Ehrenberg, A. S. C., "The Design of Replicated Studies", *The American Statistician*, 47(3): 217-228, 1993.
[28] Madachy, R. J., "System Dynamics Modelling of an Inspection-Based Process", *Proceedings of the 18th International Conference on Software Engineering*, pp. 376-386, 1996.
[29] Martin R. and Raffo, D., "Application of a Hybrid Process Simulation Model to a Software Development Project", *Journal of Systems and Software*, 59(3): 237-246, 2001.
[30] Miller, J., Roper, M., Wood, M. and Brooks, A., "Towards a Benchmark for the Evaluation of Software Testing Techniques", *Information and Software Technology*, 37(1): 5-13, 1995.
[31] Miller, J., "Replicating Software Engineering Experiments: A Poisoned Chalice or the Holy Grail", *Information and Software Technology*, 47(4): 233-244, 2005.
[32] Munson, J. C. and Khoshgoftaar, T. M., "The Detection of Fault-Prone Programs", *IEEE Transactions on Software Engineering*, 18(5): 423-433, 1992.
[33] Murugesan, S., "Attitude Towards Testing: A Key Contributor to Software Quality", *Proceedings of 1st International Conference on Software Testing, Reliability and Quality Assurance*, pp. 111-115, 1994.
[34] Musa, J., Iannino, A. and Okumoto, L., *Software Reliability Measurement, Prediction, Application*, McGraw-Hill, 1987.
[35] Musa, J. and Ackerman, A., "Quantifying Software Validation: When to Stop Testing?", *IEEE Software*, 6(3): 19-27, 1989.
[36] Ng, S. P., Murnane, T., Reed, K., Grant, D. and Chen, T. Y., "A Preliminary Survey on Software Testing Practices in Australia", *Proceedings of 2004 Australian Software Engineering Conference*, pp. 116-125, 2004.
[37] Patton, M. Q., *Qualitative Evaluation and Research Methods* (2nd ed.), Sage Publications, 1990.

[38] Pfahl, D., Laitenberger, O., Ruhe, G., Dorsch, J. and Krivobokova, T., "Evaluating the Learning Effectiveness of Using Simulations in Software Project Management Education: Results from a Twice Replicated Experiment", *Information and Software Technology*, 46(2): 127-147, 2004.

[39] Pfleeger, S. L., "Soup or Art? The Role of Evidential Force in Empirical Software Engineering", *IEEE Software*, 22(1): 66-73, 2005.

[40] Raffo, D. M. and Kellner, M. I., "Analyzing Process Improvements Using the Process Tradeoff Analysis Method", *Proceedings of Software Process Modelling and Simulation Workshop*, 2000.

[41] Robson, C., *Real World Research* (2nd ed.), Blackwell Publisher, 2002.

[42] Runeson, P., Andersson, C. and Höst, M., "Test Processes in Software Product Evolution – A Qualitative Survey on the State of Practice", *Journal of Software Maintenance and Evolution: Research and Practice*, 15(1): 41-59, 2003.

[43] Russell, G. W., "Experience with Inspection in Ultralarge-Scale Development", *IEEE Software*, 8(1): 25-31, 1991.

[44] Sheldon, F. T., Kavi, K. M., Tausworthe, R. C., Yu, J. T., Brettschneider, R. and Everett, W. W., "Reliability Measurement: From Theory to Practice", *IEEE Software*, 9(4): 13-20, 1992.

[45] Sjøberg, D. I. K., Hanny, J. E., Hansen, O., Kampenes, V. B., Karahasanović, A., Liborg, N-K. and Rekdal, A. C., "A Survey of Controlled Experiments on Software Engineering", *IEEE Transactions on Software Engineering*, 31(9): 733-753, 2005.

[46] Sommerville, I., *Software Engineering* (7th ed.), Addison-Wesley, 2004.

[47] Stikkel, G., "Dynamic Model for the System Testing Process", to appear in *Information and Software Technology*, 2006.

[48] Stringfellow, C. and Amschler Andrews, A., "An Empirical Method for Selecting Software Reliability Growth Models", *Empirical Software Engineering*, 7(4): 319-343, 2002.

[49] Tichy, W. F., Lukowitz, P., Prehelt, L. and Heinz, E. A., "Experimental Evaluation in Computer Science: A Quantitative Study", *Journal of Systems and Software*, 28(1): 9-18, 1995.

[50] Trochim, W. M. K., *The Research Methods Knowledge Base* (2nd ed.), Atomic Dog Publisher, 2001.

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

[52] Wood, A., "Predicting Software Reliability", *IEEE Computer*, 29(11): 69-78, 1996.

[53] Wood, A., "Software Reliability Growth Models: Assumptions vs. Reality", *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pp. 136-141, 1997.

[54] Zelkowitz, M. V. and Wallace, D., "Experimental Validation in Software Engineering", *Information and Software Technology*, 39(11): 735-743, 1997.

[55] Zendler, A., "A Preliminary Software Engineering Theory as Investigated by Published Experiments", *Empirical Software Engineering*, 6(2): 161-180, 2001.

PAPER 1

What Do We Know about Defect Detection Methods?

Per Runeson, Anneliese Andrews, Carina Andersson, Tomas Berling, Thomas Thelin

IEEE Software, 23(3) May/June 2006, (c) IEEE.

I

Abstract

Serious efforts are spent on defect detection during software development. Hence, it is important that we use the most efficient and effective methods for defect detection. In this paper we guide the practitioners to choose efficient and effective defect detection methods with the Evidence-Based Software Engineering (EBSE) approach. We survey existing empirical studies, that compare inspection and testing techniques. Ten experiments and two case studies are analyzed, and practical implications from the studies are derived. We recommend that inspections are used for requirements and design, while testing is used for code. As different defect detection methods find different types of defects, the methods may be complementary. We also conclude that despite extensive research, the results are often inconclusive. Finally, we list influencing factors, which help frame the question, and guide the practitioner in further steps to be taken to integrate and evaluate the detection methods in their environments.

Defect detection in development of software products requires serious efforts. Hence, it is important to use the most efficient and effective methods. Software practitioners have to decide which methods to use and for what purpose. Evidence-Based Software Engineering (EBSE) [7] is an approach to support these decisions, which involves defining relevant questions, surveying and appraising available empirical evidence, and integrating and evaluating new practices in the target environment. This paper aids in defining questions regarding defect detection techniques, and presents a survey of empirical studies on testing and inspection techniques which we interpret in terms of practical use. The final step is an invitation to practitioners, to make evidence-based decisions, to evaluate the performance of the methods in their own environment and to continuously improve.

1. Introduction

Defect detection techniques, inspections and testing, are quite well investigated in empirical studies, in isolation. Aurum et al [2] summarize 25 years of empirical research on software inspections. They survey 30+ studies, which investigate different reading techniques, team sizes, meeting gains etc. Similarly, Juristo et al [12] summarize 25 years of empirical research on software testing. They survey 20+ studies and compare testing techniques within so-called "families" as well as across families. Despite the large number of studies, they conclude, "our current testing technique knowledge is very limited".

We know even less about the relationship between inspection and testing, and the effects of combining both. Laitenberger summarizes six experiments comparing inspection and testing methods and adds one experiment on the combination of the two [14]. Later, five experiments [20][19][11][1] and two case studies [6][5] have added to the knowledge on inspections versus testing. Based on these studies, we search evidence that would help when choosing between these types of defect detection methods.

Dybå, Kitchenham and Jørgensen presented a framework for EBSE recently [7]. We follow their procedures and 1) to define more precise questions regarding defect detection methods, 2) to survey empirical studies in the field, and 3) to analyze the practical implications of the findings for defects of different origin. Finally, we invite practitioners 4)

to integrate the findings in their environments and 5) to evaluate the method's performance for continuous improvement.

2. What influences the choice of method?

The choice of detect detection method depends on factors such as the artifacts, which types of defects they contain, who is doing the detection, how it is done, for what purpose and in which activities. Factors also include which criteria rule the evaluation.

Artifact: Which is the artifact to be assessed? Requirements? Design? Code? Testing requires an executable representation, i.e. code, while inspection can be applied to any type of artifact. Most experiments, by necessity, use artificial, small artifacts. Using industrial artifacts improves the generalizability of the study, but also leads to more confounding factors.

Types of defects: What types of defects are contained in the artifacts? There is a big difference between grammatical errors in code and missing requirements in a requirements specification. The testing and inspection methods may be better or worse for different types of defects. In this paper, we firstly classify defects based on their origin: requirements, design or code. Second, many of the empirical studies categorize defects along two dimensions, as proposed by Basili [3]. Dimension one classifies defects as either an omission (something is missing) or a commission (something is incorrect), while dimension two defines defect classes according to their technical content [3][5]. Other classifications focus on the severity of the defect in terms of its impact for the user, unimportant, important and crucial [21].

Actor: Who is the reviewer or tester? A freshman student in an experimental setting or an experienced software engineer in industry? What is the incentive for doing a good job in an empirical study, which is not part of a real development project [10]? To confound matters further, experienced students have been observed to out-perform recently graduated engineers [9].

Technique: Which technique is used for inspection and testing respectively? There are many techniques for both inspection and testing. Hence, we refer to inspection and testing as families of verification techniques. For testing, specifically, we distinguish between structural testing (i.e. white-box) and functional testing (i.e. black-box).

Purpose: What is the purpose of the inspection and testing activity? The activity may contribute to validation, i.e. assuring that the correct system is developed, and to verification, i.e. assuring that the system meets its specifications. The primary goal of both inspection and testing is to find defects, but more specifically, is it to detect the presence of a defect, for later isolation by someone else, or is it to isolate the underlying fault? Testing reveals the presence of a defect through its manifestation as a failure during execution. Inspections point directly at the underlying fault. Further, there are secondary purposes for inspection and testing. Inspections may contribute to knowledge transfer, for example, while testing in a test-driven design setting produces test specifications that also may constitute a detailed design specification [4].

Defect detection activities: A specific defect may originate in one stage of the development and may be detected in the same, or a later stage. The defect is present in an artifact, for example, a missing interface in a design specification, and may propagate to the coding stage and result in missing functionality in the code. This design defect may be detected during a design inspection, code inspection/unit test, function test or system test. However, as defect detection activities are focused on a certain abstraction level, we consider primary defect detection activities are those at the same level of abstraction, i.e. design inspection and function test, whereas code inspection or testing are secondary defect detection activities for design defects.

Table 1 illustrates primary and secondary defect detection activities for defects originating from requirements, design and coding (cf. column 1). Cells numbered 1 represent primary defect detection activities, while cells numbered 2 classify secondary activities.

Table 1. Phases for origin and detection of defects

Origin of defect	Defect detection activity					
	Requirements inspection	Design inspection	Code inspection	Unit test	Function test	System test
Requirements	1	2	2	2	2	1
Design	N/A	1	2	2	1	2
Coding	N/A	N/A	1	1	2	2

Evaluation criteria: What are the criteria for selection of techniques? Is the most effective or the most efficient method to be chosen? Efficiency in this context means the number of defects found per time unit spent on verification, and effectiveness means the share of the existing defects found.

The above listed factors show that there are many variations to take into account. When searching evidence for the use of some defect detection method, specific levels of these factors must be chosen to guide the appraisal of empirical evidence.

3. Survey of empirical studies

Available sources of empirical evidence are experiments and case studies. Experiments provide good internal validity [22]. They can manipulate the context of the investigation and control many parameters. It is difficult to achieve high external validity in experiments. Case studies have a more realistic context, but are subject to confounding factors by their very nature. The generalizability of a case study depends on how similar the case study is to an actual situation. No single experiment or case study can provide a complete answer, but a collection of studies may contribute to the understanding of a phenomenon.

This survey covers experiments comprising both inspection and testing; nine experiments on code defects, one experiment on design defects and two case studies on a comprehensive defect detection process. Table 2 summarizes empirical studies involving both inspection and

Table 2. Summary of surveyed empirical studies on inspection versus testing.

Study	Type	Technique	Purpose	Artifact
Hetzel [8] 1976	Experiment		Detection	Code modules (3 PL/1 programs, 64-170 statements)
Myers [16] 1978	Experiment		Detection	Code (PL/1, 63 statements)
Basili and Selby [3] 1987	Experiment	Functional test vs. structural test and inspection	Detection	Code (Fortran, Simpl-T, 169, 145, 147 and 365 LOC)
Kamsties and Lott [13] 1995	Experiment (replication)		Detection and isolation	Code (c, 211, 248 and 230 LOC)
Roper et al [18] 1997	Experiment (replication)		Detection	Code, see Kamsties and Lott
Laitenberger [14] 1998	Experiment	Inspection followed by structural testing	Detection	Code (c, 262 LOC)
So et al [20] 2002	Experiment	Voting, testing, self-checks, code reading, data-flow analysis, Fagan inspection	Detection	Code, (8 Pascal programs, 1201-2414 LOC)
Runeson and Andrews [19] 2003	Experiment	Inspection vs. structural testing	Detection and isolation	Code (c, 190 and 208 LOC)
Juristo and Vegas [11] 2003	Experiment (replication)	Function test vs. structural test and inspection	Detection	Code, see Kamsties and Lott + one new Code, see Kamsties and Lott
Andersson et al [1] 2003	Experiment	Inspection vs. functional test	Detection	Design (text, 9 pages, 2300 words)
Conradi et al [6] 1999	Case study	Inspection, desk checks and test		Design, code
Berling and Thelin [5] 2003	Case study	Inspection, unit test, sub-system and system test	Isolation	Requirements, design, code

Table 2. Summary of surveyed empirical studies on inspection versus testing (cont.).

Study	Type of defects[a]	Actors	Result
Hetzel	-	39 students	Effect: Testing > Inspection
Myers	15	59 professionals	Effect: Inspection = Testing; Complements. Different for some classes of defects
Basili and Selby	4 programs, total 34 defects.Om/com:(0/2 ini, 4/4 cmp, 2/5 cnt, 2/11 int, 2/1 d, 0/1 cos)	32 professionals + 42 advanced students	Depending on software type
Kamsties and Lott	3 programs, 6/9/7 defects, (0/2/0 ini, 0/0/1 cmp, 3/2/3 cnt, 0/3/0 int, 2/1/2 d, 1/1/1 cos)	27 + 15 students	Effect: no; Efficiency: Testing > Inspection
Roper et al	3 programs, 8/9/8 defects	47 students	Effect: no; Efficiency: Testing > Inspection Combination better
Laitenberger	13 (2 ini, 4 cnt, 2 cmp, 2 cos, 3 d)	20 students	Not complementary
So et al	179 major faults (minor only considered in the second experiment)	26 + 15 students	1.Voting > Testing > Inspection; 2.Voting, Testing, Inspection are complementary
Runeson and Andrews	9 in each version (0/1 ini, 5/0 cnt, 1/0 int, 0/4 cmp, 1/2 cos, 1/2 d)	30 students	Test > Inspection for detection; Inspection > Test for isolation
Juristo and Vegas	4 programs, 9 defects each.Om/com:(1/2 ini, 2/2 cnt, 1/1 cos)	196 students	Effect: Different for different fault types Testing >Inspection
	3 programs, 7 defects in each. Om/com: (1/1 ini, 1/1 cnt, 0/1 cmp, 1/1 cos)	46 students	
Andersson et al	2 versions of a design document, 13/14 defects, (3/4 crucial, 5/6 important, 5/4 unimportant)	51 students	Inspection > Testing Different faults found (one version)
Conradi et al	1502 and 6300 in two studied projects respectively.	Professionals	Inspection > Testing
Berling and Thelin	244 (45 likely to propagate to code)	Professionals	Little overlap between testing and inspection

a. om=Omission; com=Comission; ini=Initialization; cmp=Computation; cnt=Control; int=Interface; d=Data; cos=Cosmetic)

testing, and Table 3 presents data from the studies. We report the outcome of the experiments and relate them to the results of the case studies conducted. Issues covered are:

- Requirements defects
- Design defects
- Code defects
- Different defect types
- Efficiency vs. effectiveness

3.1 Requirements defects

There are several experiments comparing different requirements inspection methods [2], but none comparing it to a testing method. The choice between requirements inspection and system testing is quite obvious, and needs no experiments: spending effort upfront to establish a good set of requirements is more efficient, than developing the system based on incorrect requirements and then rework it. The case study by Berling and Thelin supports this assumption [5].

3.2 Design defects

The key question regarding design defects is whether inspection of design documents or testing of the implemented function is more efficient. This issue is addressed by one experiment by Andersson et al [1], who observed defect detection in a design specification and in a log from function test execution.

The inspections were significantly more effective and efficient than testing. The variation between the experimental groups is large. More than half of the defects were found in inspection (53.5%) while slightly fewer were found in testing (41.8%) (See Table 3). The efficiency was 5 defects per hour for inspection and less than 3 per hour for testing; hence the data favor design inspections.

The analysis did not take into account rework costs. A defect detected during design inspection is much cheaper to correct than one detected in function testing, as the latter involves rework of the design and code. This

Table 3. Average values of effectiveness and efficiency for defect detection.

		Study	Inspection effectiveness[a, b]	Inspection efficiency[b, c]	Testing effectiveness[a, d]	Testing efficiency[c, d]	Different faults found
Experiments	Code	**Hetzel [8]**	37.3	-	47.7; 46.7	-	-
		Myers [16]	38.0	0.8	30.0; 36.0	1.62; 2.07	Yes
		Basili and Selby [3]	54.1	Dependent on software type	54.6; 41.2		Yes
		Kamsties and Lott [13]	43.5 50.3	2.11 1.52	47.5; 47.4 60.7; 52.8	4.69; 2.92 3.07; 1.92	Partly (for some types)
		Roper et al [18]	32.1	1.06	55.2; 57.5	2.47; 2.20	Yes
		Laitenberger [14]	38	-	9[e]	-	No
		So et al [20]	17.9; 34.6	0.16; 0.26	43.0	0.034	Yes
		Runeson and Andrews [19]	27.5	1.49	37.5	1.8	Yes
		Juristo and Vegas [11]	20.0 -	-	37.7; 35.5 75.8; 71.4	- -	Partly (for some types)
	Design	**Andersson et al [1]**	53.5	5.05	41.8	2.78	Yes for one version, no for the other
Case studies		**Conradi et al [6]**	-	0.82	-	0.013	-
		Berling and Thelin [5]	86.5	0.68 (0.13)	80	0.10	Yes

a. Percentage of the defects in the artifact that are detected. In the case study, this is based on estimated number of defects

b. For multiple figures, they are reported in this order: Code reading, Fagan inspection

c. Detected defects per hour

d. For multiple figures, they are reported in this order: Functional Test; Structural Test.

e. Note that the testing is conducted in sequence after the inspection.

implies even greater efficiency for design inspections compared to functional testing.

These results are confirmed by an industrial case study by Berling and Thelin [5]. They studied five incremental project iterations. Inspections detected on average 0.68 defects per hour, while testing detected 0.10 defects per hour (see Table 3). For the fraction of faults that was estimated to propagate into code, the rate was 0.13 defects per hour. This case study also reports slightly higher effectiveness for inspections, but the differences are small. Similar results are found in a case study by Conradi et al [6]. 0.82 defects per hour were detected in design inspection, while only 0.013 defects per hour were detected in function test. Hence, the empirical data support design inspections as a more efficient means for detecting design defects.

3.3 Code defects

Code defects are investigated in the largest number of experiments. However, there is no clear answer whether code inspection or functional/ structural testing is preferable.

Miller [15] attempted to analyze the results from the first five studies in Table 2. However, the variation among the studies was too large to apply meta-analysis techniques. Hence, we simply rank the techniques with respect to their effectiveness and efficiency. Figure 1 presents the effectiveness, i.e. the percentage of the defects found by the different techniques.

Figure 1 reveals that none of the techniques emerges as a clear winner or loser. All three techniques are ranked most effective in at least one study and least effective in at least one study. However, in all cases where three techniques are compared, the difference between the average effectiveness of the first and second technique, is smaller than between the second and third. The data does not allow a scientific conclusion as to which technique is superior, but from a practical perspective it seems that testing is more effective than code inspections.

Figure 2 presents the efficiency ranking, i.e. number of found defects per time unit, for those studies reporting efficiency numbers. For the others, the space is left blank. The efficiency ranking differs from the effectiveness ranking. The study by Myers [16] ranked inspection most effective and least efficient, while the study by So ranked testing most effective and least efficient. The study by Roper et al [18] ranked

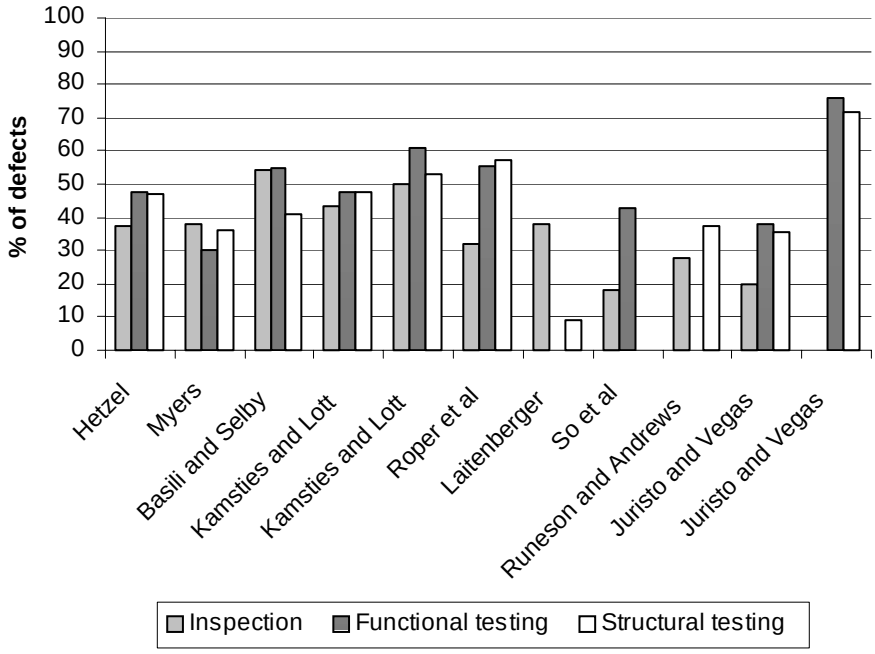

Figure 1. *Average* effectiveness *of techniques for code defect detection*

functional testing most effective and structural testing most efficient, but the differences are small.

In the studies by Kamsties and Lott [13], and Runeson and Andrews [19], they distinguish between detection and isolation of defects. In the former study, this moved inspections to the second rank. In the latter study, this distinction did not change the rank between the two studied techniques. The study by Roper et al [18] also distinguishes between detection and isolation, which improved the performance of the inspection, but it still ranked third.

In summary, the results regarding code inspection or functional/ structural testing are inconclusive, although they tend to favor testing. These studies are, however, conducted in isolation, and thus do not take secondary issues into account, such as information spreading effects of inspections, value of unit test automation, cost and intrinsic values of a test suite or test driven design.

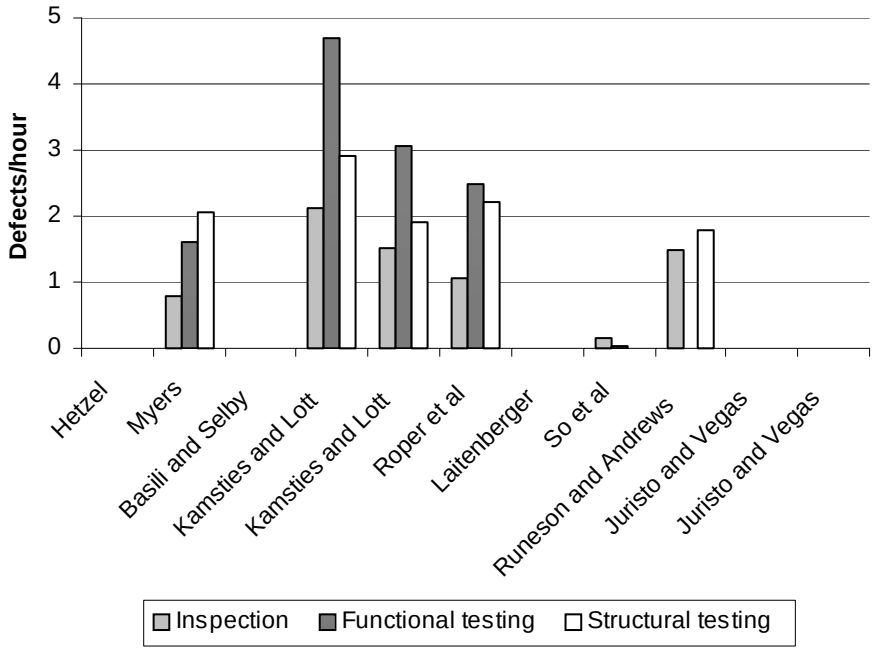

Figure 2. *Average* efficiency *for techniques for code defect detection.*

3.4 Different defect types

The studies vary widely in the types of defects the artifacts contain. If that is indeed an important factor, then the absence or presence of different types of defects may affect efficiency and effectiveness. Table 2, column 5 shows the types of defects in each study, if they are known.

It is difficult to compare the studies. First, only six of the studies report defects by type and can be used to investigate whether the differences depend on the fault type, not only on the technique. Second, the frequencies of the different defect types vary widely among the remaining studies. Third, only a fraction of defects are found and thus, while we may know the defect type frequencies for each artifact, we may not know the defect type frequencies of the defects that were found. Given the low proportion of defects found, this makes cross-study comparison difficult, if not impossible. Fourth, classification schemes involve a subjective judgment that may confuse the results of the classification as such.

Few studies investigate the relationship between defect types and inspection versus testing. Runeson and Andrews [19] found that there was a statistically significant difference between defect types found by inspection and testing, respectively. This implies that the techniques' performance is sensitive to the defect type. The same study indicates that the subjects tend to prefer testing over inspection, and this is a piece of qualitative information that should be taken into account.

Kamsties and Lott [13] found no difference in performance between defect types. Juristo and Vegas [11] replicated this experiment twice. In the first replication, cosmetic faults were most difficult to find, irrespective of technique. However, code inspection was not affected by defect type, contradicting an earlier study by Basili and Selby [3]. Functional testing performed better than structural testing for faults of omission, and testing techniques performed better than inspection. They could not detect a clear pattern as to which defect types are more easily detected by which technique. In summary, the jury is still out as to the effect of defect types on the performance of inspection versus testing.

3.5 Effectiveness and efficiency

Absolute levels of effectiveness of defect detection techniques are remarkably low. In all but one of the experimental studies, the subjects found only between 25 to 50% of the defects on average during inspection, and slightly more during testing (30-60%). This means that on average, more than half the defects remain! The effectiveness figures reported in the Berling case study [5] are 86.5% for inspections and 80% for testing. However, these are based on estimated number of defects that could possibly be found by the technique; not on all defects existing in the documents.

In the experimental studies, 1-2 defects are found per hour, 2.5 defects per hour at most. The size of the artifacts studied in experiments is at most a few hundred lines of code. This is small from an industrial perspective where professionals deal with more complex artifacts, struggle with more communication overhead etc. Consequently, the efficiency in the industrial case studies is lower; 0.01-0.82 defects per hour. The variation is also much larger, which may be due to different company measures of efficiency [5].

The practical implication of the low effectiveness and efficiency values of the primary defect detection methods, is that the secondary detection

methods may play a larger role than is accounted for in the empirical studies we surveyed.

3.6 Validity of the empirical results

The studies we summarized show indications of pros and cons for the two families of defect detection techniques, but no clear winner. But how valid are the results?

Threats to validity of empirical studies can be analyzed along four dimensions: internal, conclusion, construct and external validity [22]. From a practitioner point of view, external validity ranks first, as indicated by Rainer et al [17]. From a researcher point of view, internal validity is traditionally considered the key to successful research. However, it is important to balance all dimensions of validity to achieve trustworthy empirical studies.

The internal validity of the surveyed experimental studies seems quite high. The studies are conducted by established researchers, mostly in student environments with experienced students. On the other hand, the inconclusive results indicate that there are factors at work that are not under experimental control. The studies also seem to control conclusion validity threats, as established analysis methods are used.

For case studies, on the other hand, it is harder to achieve high internal and conclusion validity. Confounding factors may interact with the factor under study and threaten internal validity. Because data collection often is defined for purposes other than the empirical study, conclusion validity may be threatened.

Construct validity is potentially threatened in both experiments and case studies, since different instantiations of defect detection methods are used to represent test and inspection techniques. The studies listed in Table 2 present large variations within the families of techniques as well as between the families, both in experiments and case studies.

The major threat for experiments is the external validity as they are conducted on small artifacts, mostly with students as subjects. The case studies suffer less from external threats, although there is a risk that the conditions specific to a particular case greatly influence the outcome.

4. What is the answer?

Our analysis of existing empirical studies showed no clear cut answer to the question which defect detection method to choose. However, from a practical viewpoint, some tendencies can be observed:

- For requirements, there is no empirical evidence at all, but the fact that costs for requirements inspections are low compared to implementing incorrect requirements indicates that requirements defects should be found through inspection.
- For design specifications, one experiment indicates support for inspections before functional testing both with respect to efficiency and effectiveness. Case studies in industry indicate the same.
- For code, functional or structural testing is ranked higher than inspection in a majority of the studies. Some studies conclude testing and inspection find different kinds of defects, hence being complementary. Results differ when studying isolation of a fault, not only detection of a defect.
- The effectiveness of verification activities is low; only 25-50% of the defects in an artifact are found using inspection, and 30-60% using testing. This makes secondary defect detection activities important. The efficiency is in the magnitude of 1-2 defects per hour spent on inspection or testing.

Several papers call for “replication of this study” and “further work on this topic”. Some experiments are actually replications of each other, and many replications do not corroborate the original studies. It seems that factors are at work that are not measured or controlled, but which nonetheless influence the performance of the defect detection methods. It may be useful to investigate other, “softer” factors, for example, motivation and satisfaction [10], and in particular apply methods in practice, monitor and follow up.

5. Which detection method should I choose?

For the practitioner seeking an answer to the question "which defect detection method should I choose?", the empirical results we analyzed can be used in two different ways. Either they can be used as a guideline for a high level strategy of defect detection methods, or they can be used in a full EBSE fashion [7], finding an answer to a specific question.

In strategy definitions, the summary of the empirical studies can act as a general guideline for which defect detection methods should be used for different purposes and at different stages of development. The summarized findings are not novel, nor are they strictly empirically based. However, the definition of such a strategy would benefit many organizations and projects. Making the defect detection strategy explicit helps communicating values internally, making the project members aware of how their work fits into the complete picture. The strategy could also be a starting point for EBSE-based investigations of more specific trade-offs between different methods.

Applying a full EBSE cycle [7], the variation factors outlined in Section 2 can be used to precisely specify the question of which defect detection technique to use. In Table 4, an example is shown of levels of variation factors in the search for a feasible defect detection technique.

The factors help framing the general question into the more precise one: "Which defect detection technique should we use to detect as many critical and important omission defects in interfaces as possible, in code inspection and unit testing, conducted by novice testers and

Table 4. Example definition of factor level

Factor	Level
Artifact	Code modules
Types of defects	Omitted, crucial and important interfaces
Actor	Novice testers and programmers
Technique	Any feasible technique
Purpose	Design verification
Defect detection activity	Code inspection and unit testing
Evaluation criteria	Effectiveness

programmers?". Then the studies in Table 2 can be surveyed in detail, to serve as a basis for the decision. Hypothesize that a combination of structural and functional testing is chosen. Then previous experience from the organization must be taken into account to anchor the decision. As a final step in the cycle, the performance of the selected technique must be monitored and evaluated.

6. Summary

Software practitioners face the question of which defect detection technique to use in a specific context. Evidence-Based Software Engineering (EBSE) may help with this selection. In this paper we list influencing factors, which help frame the question, we survey available empirical evidence, and provide guidance for the steps to be taken by practitioners. Thereby, a more informed decision may be made. Further, if the industrial evaluation is fed back to the research community, the body of knowledge will increase of the practical utility of the methods.

7. References

[1] Andersson, C., Thelin, T., Runeson, P. and Dzamashvili, N., "An Experimental Evaluation of Inspection and Testing for Detection of Design Faults", *Proceedings IEEE/ACM International Symposium on Empirical Software Engineering*, pp. 174-184, 2003.

[2] Aurum, A., Petersson, H. and Wohlin, C., "State-of-the-art: Software Inspections after 25 Years", *Software Testing, Verification and Reliability*, 12(3):133-154, 2002.

[3] Basili, V. R. and Selby, R., "Comparing the Effectiveness of Software Testing Strategies", *IEEE Transactions on Software Engineering*, 12(12):1278-1296, 1987.

[4] Beck, K., *Test Driven Development: By Example*, Addison-Wesley Professional, 2002.

[5] Berling, T. and Thelin, T., "An Industrial Case Study of the Verification and Validation Activities", *Proceedings 9th International Software Metrics Symposium*, pp. 226-238, 2003.

[6] Conradi, R., Marjara, A. S. and Skåtevik, B., "An Empirical Study of Inspection and Testing Data at Ericsson, Norway", *Proceedings 24th NASA Software Engineering Workshop*, Greenbelt/Washington, USA, 1-2 Dec. 1999.

[7] Dybå, T., Kitchenham B. A. and Jørgensen, M., "Evidence-Based Software Engineering for Practitioners", *IEEE Software*, 22(1):58-65, 2005.

[8] Hetzel, W. C., *An Experimental Analysis of Program Verification Methods*, Ph.D. dissertation, University of North Carolina, Chapel Hill, 1976.

[9] Höst, M., Regnell, B. and Wohlin, C., "Using Students as Subjects - A Comparative Study of Students and Professionals in Lead-Time Impact Assessment", *Empirical Software Engineering*, 5(3): 201-214, 2000.

[10] Höst, M., Wohlin, C. and Thelin, T., "Experimental Context Classification: Incentives and Experience of Subjects", *Proceedings of the 27th International Conference on Software Engineering*, pp. 470-478, 2005.

[11] Juristo, N. and Vegas, S., "Functional Testing, Structural Testing and Code Reading: What Fault Type Do They Each Detect?", In *Empirical Methods and Studies in Software Engineering*, R. Conradi and A. I. Wang (Eds.), Springer, pp. 208-232, 2003.

[12] Juristo, N., Moreno, A. M. and Vegas, S., "Reviewing 25 Years of Testing Technique Experiments", *Empirical Software Engineering*, 9(1-2): 7-44, 2004.

[13] Kamsties, E. and Lott, C. M., "An Empirical Evaluation of Three Defect-Detection Techniques", *Proceedings of the 5th European Software Engineering Conference*, pp. 362-383, 1995.

[14] Laitenberger, O., "Studying the Effects of Code Inspection and Structural Testing on Software Quality", *Proceedings of the 9th International Symposium on Software Reliability Engineering*, pp. 237-246, 1998.

[15] Miller, J., "Applying Meta-Analytical Procedures to Software Engineering Experiments", *Journal of Systems and Software*, 54(1):29-39, 2000.

[16] Myers, G. J., "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections", *Communications of the ACM*, 21(9):760-768, 1978.

[17] Rainer, A., Hall, T.,and Baddoo, N., "Persuading Developers to 'Buy into' Software Process Improvement: Local Opinion and Empirical Evidence", *Proceedings International Symposium on Empirical Software Engineering*, pp. 326-355, 2003.

[18] Roper, M., Wood, M. and Miller, J., "An Empirical Evaluation of Defect Detection Techniques", *Information and Software Technology*, 39(11):763-775, 1997.

[19] Runeson, P. and Andrews, A., "Detection or Isolation of Defects? An Experimental Comparison of Unit Testing and Code Inspection", *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pp. 3-13, 2003.

[20] So, S. S., Cha, S. D., Shimeall, T. S. and Kwon, Y. R.., "An Empirical Evaluation of Six Methods to Detect Faults in Software", *Software Testing, Verification and Reliability*, 12(3):155-171, 2002.

[21] Thelin, T., Runeson, R. and Wohlin, C., "Prioritized use Cases as a Vehicle for Software Inspections", *IEEE Software,* 20(4):30-33, 2003.

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

PAPER II

An Experimental Evaluation of Inspection and Testing for Detection of Design Faults

Carina Andersson, Thomas Thelin, Per Runeson, Nina Dzamashvili

Proceedings of the 2nd International Symposium on Empirical Software Engineering, pp. 174-184, 2003.

II

Abstract

The two most common strategies for verification and validation, inspection and testing, are in a controlled experiment evaluated in terms of their fault detection capabilities. These two techniques are in previous work compared applied to code. In order to compare the efficiency and effectiveness of these techniques on a higher abstraction level than code, this experiment investigates inspection of design documents and testing of the corresponding program, to detect faults originating from the design document. Usage-based reading (UBR) and usage-based testing (UBT) were chosen for inspections and testing, respectively. These techniques provide similar aid to the reviewers as to the testers. The purpose of both fault detection techniques is to focus the inspection and testing from a user's viewpoint. The experiment was conducted with 51 Master's students in a two-factor blocked design; each student applied each technique once, each application on different versions of the same program. The two versions contained different sets of faults, including 13 and 14 faults, respectively. The general results from this study show that when the two groups of subjects are combined, the efficiency and effectiveness are significantly higher for usage-based reading and that testing tends to require more learning. Rework is not taken into account,

thus the experiment indicates strong support for design inspection over testing.

1. Introduction

Verification and validation are conducted to detect faults throughout the development of a software product. The process of verification and validation takes a large share of the development cost in a software project. Verification aims at checking that the system as a whole works according to its specifications and validation aims at checking that the system behaves according to the customers' intentions. The main types of activities for verification and validation are inspections [6] and testing [7]. Software testing cannot be conducted until the software product is implemented; hence it is conducted in the later phases of software development. Since faults need to be found early to avoid costly rework, software inspections may be conducted before the product has been implemented.

Inspection and testing are the most common techniques for fault detection in software artefacts, and several empirical studies have investigated these techniques [1][11]. In order to compare inspections and testing, industrial case studies [4][5], as well as experiments [13] have been conducted. However, the experiments compared the effectiveness of testing and inspection of code, i.e. the subjects of these experiments applied inspection or testing on the same software artefacts [2][12][13][16]. These experiments focus on comparing code inspections with functional and structural testing. The results are summarized by Laitenberger [13], and suggest that several fault detection techniques should be applied to achieve software of high quality.

Reading techniques for software inspections have been evaluated empirically and several improvements have been proposed [1]. Reading techniques have been proposed and evaluated [23], for example, perspective-based reading [3]. Another promising reading technique is usage-based reading (UBR) [21], which has been empirically evaluated in three studies [22]. It is concluded that UBR is an effective and efficient reading technique. UBR focuses on the user's viewpoint, much in the same way as usage-based testing (UBT) [14], using use cases as the guide for reviewers.

Several testing techniques have been empirically evaluated [11][15] and also compared with inspections [2][20]. UBT is one of these test techniques, which focuses on the users' needs with the main purpose to estimate the reliability of the software [17]. The information used in UBT stems from the intended usage of software, and different usage profiles may be designed. In this study, the information used as input for UBT is translated from use cases. The use cases represent the intended usage of the software and are easily translated into test cases and thus provide usage information to UBT.

Since fault detection techniques are effort consuming, knowledge of how to combine inspection and testing is important. Hence, the aim of this study is to investigate software inspections (applying UBR) and testing (applying UBT), on a higher abstraction level than code in software engineering. In order to investigate this, a controlled experiment was conducted to study the effects of UBR on a design document and UBT on the corresponding programs, i.e. the faults that exist in the design documents have propagated into the code. The general research question addressed in this paper is:

- What is the impact of the two fault detection techniques (UBR and UBT)?
  – The goal of the experiment is to investigate how many faults and which faults the fault detection techniques find.

The main result of the experiment indicates that inspections of design documents are significantly more effective and efficient than trying to find the faults in the test phase. In particular if the rework costs are considered, the potential gains are larger with the inspection technique. This result confirms related research in the area of verification and validation, although most other research is focused on the code instead of the design.

The remainder of this paper is structured as follows: The two techniques used in the experiment for inspection and testing, usage-based reading and usage-based testing, are described in Section 2. In Section 3 the planning of the experiment and its pilot study is explained in detail, while we in Section 4 present how the experiment was conducted. In Section 5 the experiment results are presented. In Section 6 the results and findings, and lessons learned are discussed, and also related to previous work in this area. Section 7 gives a summary and conclusions of this study.

2. Fault detection techniques

Software testing and inspections have the same main goal, which is to detect faults. These methods are the main corrective strategies in software development and are used to increase the quality of the software products. Much research has been conducted isolated in these areas. There are only few empirical studies investigating the trade-off between inspections and testing, and how these corrective methods complement each other in the best possible way [13].

Although both fault detection techniques are used to find faults, a significant difference exists between inspection and testing. When a tester observes an anomalous behavior, a *failure* has been detected and then the tester needs to isolate the fault causing the failure [8]. On the other hand, when a reviewer detects a *fault* in the document, no isolation is needed since the root of cause is already detected. In this paper, these definitions of faults and failures are used to denote observation of test result (failures) and inspection result (faults), respectively.

The fault detection techniques used in this paper are both focused on finding faults, critical for the usage of the software system. UBR is used as the reading techniques in the preparation phase of the inspection and UBT is used as the testing technique. The aim of these methods is to focus on the users' needs throughout the development. Hence, UBR and

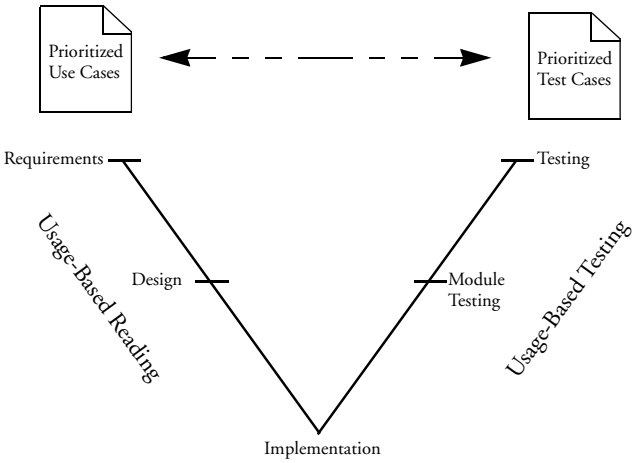

Figure 1. *The connection between the fault detection techniques compared in the empirical study.*

UBT are two complementary fault detection techniques in the software development. The relationship between UBR and UBT is shown in Figure 1.

Both these methods are focused on detecting the most critical faults from a user's viewpoint. UBR provides reviewers with prioritized use cases, and UBT provides testers with prioritized test cases. During an inspection, reviewers manually "execute" the use cases and thereby find faults. During testing, the test cases are executed on the implemented code to detect failures. Before utilizing UBR and UBT, three activities have to be conducted:

- Development of use cases – Use cases are preferably developed in the beginning of a software project. These use cases can then be used for all inspection activities in the same project. However, if use cases are not used, it is shown that it is only necessary to develop the title and purpose of the use cases in order to utilize UBR [22].
- Prioritization of use case – The use cases should be prioritized before the inspection is conducted. The prioritization can be performed pair-wise comparison by a user, group of users or some person who represents and understands the users needs.
- Translation of use cases into test cases – The test cases need to be translated from natural language to the test language used. This means that the same information is used for inspecting as well as testing.

After the prioritization of the use/test cases, UBR and UBT are performed in four basic steps, described in detail by Thelin et al. [24]:

1. Start with the use/test case with the highest priority.
2. Check the software according the use case (UBR) or the output from the testing (UBT).
3. Ensure that the software artefact fulfils the goal of the use/test case, and report the issues found.
4. Select the next use/test case and repeat from 2 until the time is up, or all use/test cases are covered.

UBT is focused on the user, much in the same way as UBR. UBT was developed before UBR with the purpose to focus on the users and to

estimate the reliability [14][17]. In fact, UBR was developed based on UBT. In this study, the prioritized use cases were used to develop the test cases. Hence, the input information to the fault detection techniques was equal.

3. Experiment planning

The planning of the experiment involved defining the hypotheses to be tested (Section 3.1), and setting up the experiment artefacts (Section 3.2). The planning also involved a pilot study (Section 3.3), selecting the subjects to participate in the main study (Section 3.4), and choosing an appropriate experimental design and defining variables and analysis methods (Section 3.5). Threats to validity have been considered before and after the experiment (Section 3.6).

3.1 Purpose and hypotheses

The purpose of the experiment is defined as follows:

Analyze the detection of design faults using inspection and testing,
for the purpose of evaluation,
with respect to their effectiveness and efficiency,
from the point of view of researchers,
in the context of Master's students, and a scaled-down system from a real application domain.

Specifically, we want to compare the use of Usage-Based Reading [21] and Usage-Based Testing [14] as presented in Section 2. UBR is a technique that requires the reviewers to go deeper into the design and really understand it, compared to UBT, which concentrates on the input-output. Hence, we expect UBR to be at least as effective and efficient as UBT in detecting the faults. In other words, the null hypotheses is that there are no differences between the techniques:

- $H_{0\ Eff}$ – There is no difference in *efficiency* (i.e. found faults per hour) between the reviewers applying UBR and the testers applying UBT.
- $H_{0\ Rate}$ – There is no difference in *effectiveness* (i.e. rate of faults found) between the reviewers applying UBR and the testers applying UBT.

- $H_{0\ Faults}$ – There is no difference in faults found, i.e. the reviewers applying UBR do not find different faults than the testers applying UBT.

The alternative hypotheses are defined in favour of the UBR, i.e.:

- $H_{a\ Eff}$ – Reviewers applying UBR are more *efficient* than testers applying UBT.
- $H_{a\ Rate}$ – Reviewers applying UBR are more *effective* than testers applying UBT.
- $H_{a\ Faults}$ – Reviewers applying UBR and testers applying UBT find different faults.

In order to compare relevant constructs, we do not include development or rework time in the data collection. This would give better credit for the UBR technique, as it is applied earlier in the development cycle, but since we only have anecdotal data on the rework effort needed, this is discussed outside the experiment analysis. In the UBR case, we measure the time spent and faults found by reading a design specification guided by a set of prioritized use cases. In the UBT case, we measure the time spent and failures found by analyzing the output traces from the execution of a set of functional test cases. The usage scenarios of the test cases are the same as those of the use cases in UBR.

3.2 Experiment artefacts

The experiment is based on material originally developed for a verification and validation course in software engineering at Lund University in Sweden [21]. The artefacts are reviewed and the code is also tested to obtain as high quality as possible. The application domain is a management system for a taxi fleet, including functionality for managing customer orders and dispatching them to an available taxi. The experiment material consists of five documents in structured text, see Figure 2: one requirements document, one design document (9 pages, 2300 words), one use case document with 12 use cases, one set of 12 test cases and finally the corresponding set of test output traces. These are presented in the form of MSC diagrams (Message Sequence Charts) [9], and contain the interaction between the user and the taxi management system.

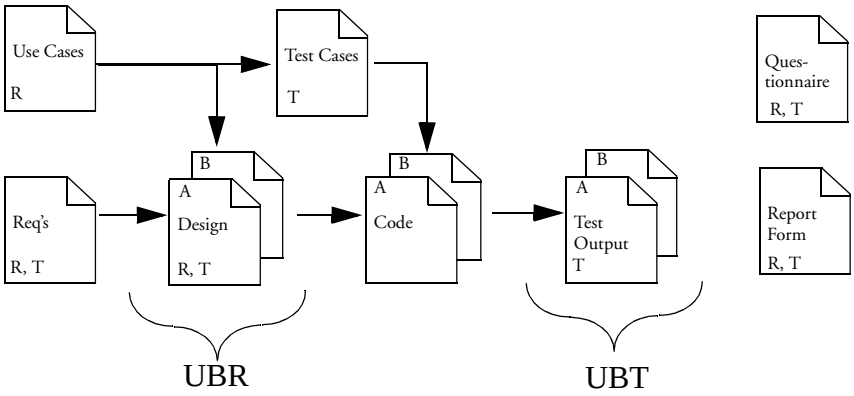

Figure 2. *Overview of the experiment artefacts. R means used by UBR and T means used by UBT. A and B refer to the different versions of the artefacts.*

The requirements document, the use case document and the design document were used in previous experiments at Lund University [22]. The test cases and the test output traces were developed for this experiment. There is also code for the implemented system, but it is not presented to the subjects. Only the outputs from the tests of the code were available to the subjects in the test output traces.The system and its artefacts are available in two versions, referred to as version A and B, with 13 and 14 faults, respectively, which give rise to corresponding failures which are possible to observe in the test output traces. The two sets of faults are analyzed and considered to have the same distribution of faults regarding type and severity. The faults were selected among the 38 faults in the design document used in earlier experiments [22]. In addition, three and four new faults of similar types were inserted in the versions respectively to achieve two different sets of artefacts. The faults in the design document were selected so that they probably might remain in a system after coding, if no design inspections were performed. Hence, the same faults that appear in the design document are causing the failures in the test output traces. Five of the faults in version A have the same originating error as five faults in version B, i.e. the two version have an intersection equal to these five faults, although they appear differently in the design and the test output traces.

In addition to the system documents, a questionnaire for self-assessment was used to collect data about the experience of the subjects, and report forms for inspection and testing for time and fault reporting

was handed out. For more details on the artefacts and the system, refer to [21].

3.3 Pilot study

A pilot study was conducted in November 2002, with 25 students of their third year of software engineering Bachelor's programme at Lund University. A smaller version of the experiment was carried out to evaluate the experiment procedures, like methods of data collection, to ensure that the questions in the self-assessment questionnaire were understandable and unambiguous, and that the introduction given before the experiment was adequate. The study was organized over one day, with an introductory part of the fault detection techniques of approximately 20 minutes and an experiment session of 3 hours. The students had participated in previous courses working with taxi management systems and had good domain knowledge. Therefore, no general introduction to the system and its documents was given. The self-assessment was delivered and filled out by each participant a few weeks before the experiment session and was analyzed to capture each student's knowledge and experience. Since no major differences were found, the students were thereafter randomly divided into two groups, each group having one treatment, UBR or UBT, to apply during the experiment.

The analysis of the pilot study showed that the introduction of the fault detection techniques may have been more focused on the inspection technique, while the students applying the testing technique asked more questions during the experiment session. During the real experiment, which was conducted a month later, the introduction and practice of the detection techniques were separated, i.e. one introduction hour was given for each technique. The self-assessment questionnaire was also revised after the pilot study to avoid ambiguity, by the means of generally decreasing the answer alternatives from 5 to 3. Only one version of the system was used in the pilot study, containing 10 faults. The analysis of these faults compelled a change and revision of some faults, in order to get faults that in a natural manner could be injected into the code from the design without being detected during implementation. The results from the pilot study were not further analyzed.

3.4 Subjects

The real experiment was conducted a month after the pilot study. In this, the main experiment, 51 fourth-year Master's level students at Blekinge Institute of Technology in Sweden participated as subjects. The experiment was included as a training part of another course in software verification and validation.

The course is a part of the Master's program in software engineering and focuses on aspects of software inspections, software testing, software reliability and analysis as well as empirical methods that can be used to evaluate software processes and new technologies. The educational purpose of the experiment was to provide the students with an opportunity to try out UBR and UBT, as well as to demonstrate how empirical methods can be used to evaluate and compare the two different techniques.

The students are considered to be rather experienced in software engineering. As part of their education they have received extensive theoretical and practical training in the software engineering domain. The students have participated in a series of software engineering projects, which are run in cooperation with industry. The aim of the projects is to simulate the challenges that are typical for software projects in industry. The projects are complex and require advanced technical skills. As a result of the project work, the students are motivated to deliver a software system according to customers' requirements and with a focus on software quality. This provides a basis to assume that the experience of the students can be considered similar to fresh software engineers working in industry.

3.5 Design

The experiment applied a two-factor blocked design [10], combining two fault detection techniques and two versions of a system and its artefacts (referred to as versions A and B). The single difference between the versions, were that they contained different sets of faults. The design yielded two groups, and every subject applied each detection technique once, but with a different version for each occasion, as shown in Table 1. To avoid any order effects, the two groups were assigned to different orders of applying the techniques. A practical constraint aroused, after a version had been used in a session in the experiment. There was a risk that the faults were made public and other subjects may have access to them.

For this reason, the first session of the experiment used only version A, and the second session, version B.

Variables: The experimental design has the *independent variables* of the two fault detection techniques, UBR and UBT, and the two versions of the program, version A and version B. The experience of the students is the *controlled variable*, while several *dependent variables* were examined and the following measures were collected: the number of faults detected by each subject, number of subjects that found each fault, and the time used for preparation and fault detection.

Table 1. Two-factor blocked design.

	UBR	**UBT**
Ver A	Group 1	Group 2
Ver B	Group 2	Group 1

Analysis methods: We analyzed the experiment data with descriptive statistics and statistical tests. The significance level for rejecting the hypotheses was set to 0.05 for all tests. Since the collected data did not follow any normal distribution we applied the Wilcoxon signed-rank test, a nonparametric equivalent to the paired two-group t-test, to evaluate the efficiency and effectiveness (H_{Eff} and H_{Rate}) for each group of subjects [19]. Efficiency is defined as the number of faults found divided by the total time spent and effectiveness (fault rate) is defined as faults found divided by the total number of existing faults. A chi-square test was used to test H_{Fault} [19].

3.6 Threats to validity

When conducting an experiment, there are always threats to the validity of the results. Depending on the purpose of the experiment, some threats are more critical than others. The purpose of the current study is to compare two approaches for fault detection, i.e. inspection and test, and the focus is the relation between the two, not on generalization. Hence, the threats to internal and construct validity are the most critical. When trying to generalize the results to another domain, the external validity becomes more important [25].

The threats to *conclusion validity* are considered small. Robust statistical techniques are used, measures and treatment implementation are considered reliable. Similar, or the same instruments are used in several experiments and the specific instances of the instruments were tried out in the pilot study. The two sessions of the experiment were conducted on adjacent days, thus reducing the risk of knowledge spreading between subjects. In addition, one version of the documents was used on the first day and the other was used the second to prevent knowledge about the specific faults to spread. The subjects were randomly assigned to groups and it was checked that the groups were balanced in terms of experience and skills via the self-assessment questionnaire.

Concerning *internal validity*, there is a limited risk of rivalry between groups since both groups applied both techniques to different artefacts. The maturity effect, i.e. that the subjects performed differently when they have gained some experience in the techniques, is possible to analyze, since both groups applied both techniques in different orders. Only one subject in each group dropped out after the first day, hence the mortality rate is low. 13 subjects did not complete at least one of the given tasks during the experiment sessions and these data points are excluded from the analysis. There is a risk that the five identical faults, which existed in both program versions, is not reported during the second day. The students were told that there should not exist the same faults in both versions, so they have probably not been searching for these faults specifically. However this is considered as a small risk, since the faults appear in different forms depending on detection technique.

Threats to *construct validity* are reduced by having the use cases and the test cases based on the same scenarios. Thus, both groups have the same information, although in different forms. Further, both groups have access to the same requirements specification. The threat to the construct validity is for the testing method that it is paper-based, rather than computer-based as testing normally is, thus making the testing less dynamic. On the other hand, making the testing computer-based require more training to get into the test environment.

Concerning *external validity*, the use of students as subjects is a threat. However, the students are fourth year Master's students in software engineering, hence more representative than freshmen students [18]. Additional threats to the external validity are the artefacts used in the experiment, which are in the smaller range for a real-world problem, even though they describe a real-world problem. For the results to be valid in a

real world setting it is also required that the design documents during the following implementation not are exposed to change management.

4. Operation

The experiment was conducted in December 2002 and was organized in two sessions over two days. The first day started with a general introduction to the Taxi management system. According to the self-assessment, which was conducted a few weeks earlier, the students' skills and previous experience did not differ very much; therefore the students were randomly divided into two groups, referred to as groups 1 and 2. Each group attended an introduction and practice for the fault detection technique they were going to use the first day. During the practice was the fault detection technique applied on a minor system. The experiment session was conducted during the afternoon, with the subjects conducting the inspection and testing individually. Though, the subjects were during the experiment sessions free to take breaks whenever they wanted. However, they were asked to not discuss the detection techniques and the faults they had found. The second day the groups received the introduction and practice in the technique they had not used the day

Table 2. Schedule for the experiment.

Time		Group 1	Group 2
Day 1 (10.15 a.m. - 11.00 a.m.)	45 min.	General introduction to the Taxi Management System	
Day 1 (11.15 a.m. - 12.00 a.m.)	45 min.	Introduction to UBR	Introduction to UBT
Day 1 (13.15 p.m. - 17.00 p.m.)	3 h 45 min.	Preparation and Fault Detection	
Day 2 (9.15 a.m. - 10.00 a.m.)	45 min.	Introduction to UBT	Introduction to UBR
Day 2 (10.15 a.m. - 13.00 p.m.)	2 h 45 min.	Preparation and Fault Detection	

before, and took afterwards part in the second day's experiment session, see schedule in Table 2. The subjects were told that they could leave the room when they considered themselves finished with the assigned tasks, or when the time was up. However, none was under time pressure during the first day's session. The second day, the students were more acquainted with the documents and one hour less was planned for the experiment session.

5. Analysis and results

This section examines the findings of the experiment. To analyze the impact of the two techniques, two sets of data were collected: the faults found and the time spent. The evaluation of the subjects' reported faults, with respect to whether it was a fault or not, were conducted by the researchers. Any false positives (reported issues that are not in fact faults) were ignored in the subsequent analysis.

5.1 Preparation and fault detection time

The subjects logged the time for preparation and fault detection time. During preparation time they read though the documents. The mean time and standard deviation values are presented in Table 3. Generally the students used more time, when applying UBT, both for preparation and fault detection, compared to applying UBR. During the second day the students were more acquainted to the documents and the system, and less time was used for both techniques, both in preparation and fault

Table 3. Preparation and fault detection times (minutes).

		Version A		Version B	
	Technique	**UBR**	**UBT**	**UBR**	**UBT**
mean	preparation	18.5	22.2	5.6	9.3
	fault detection	95.3	118.6	67.9	82.0
	total	113.8	140.8	73.5	91.3
std. dev.	preparation	8.4	10.2	4.2	6.8
	fault detection	20.4	21.4	16.2	27.3
	total	24.4	24.9	14.8	28.7

detection. Both treatment groups were using approximately 35% less total time the second day compared to the first day.

5.2 Efficiency and effectiveness

Values of efficiency and effectiveness for each technique were calculated as described in Section 3.5. In the Wilcoxon signed-rank test, subjects that had not completed the given tasks in one of the treatments were excluded from this analysis, which gave a total of 18 subjects in group 1 and 20 subjects in group 2.

Firstly, the results for efficiency and effectiveness were analyzed without concern to other factors like the different program versions and the two groups. The box plots in Figure 3 show efficiency for UBR and UBT, with better results for the subjects applying UBR. Figure 4 shows the values of effectiveness for the two techniques, also with better results for subjects applying UBR. The statistical significance of these results were tested and show that both for efficiency and effectiveness are statistical significant difference obtained, with p-values <0.001 and 0.010, respectively.

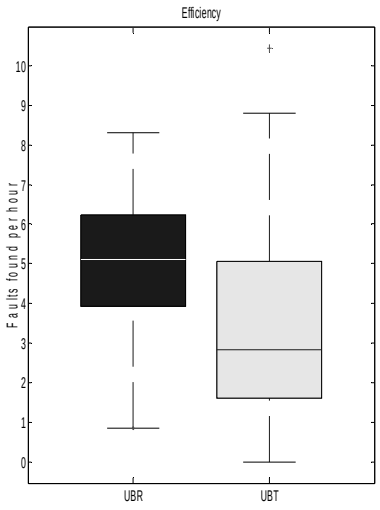

Figure 3. *Efficiency for UBR and UBT.*

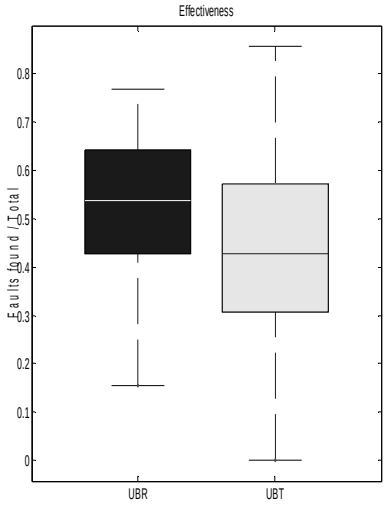

Figure 4. *Effectiveness for UBR and UBT.*

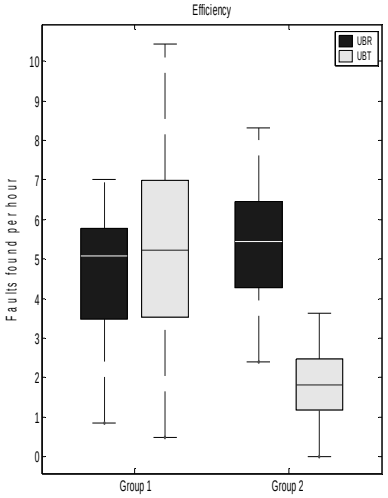

Figure 5. *Efficiency for group 1 and 2. The black box plots show UBR, while the grey show UBT.*

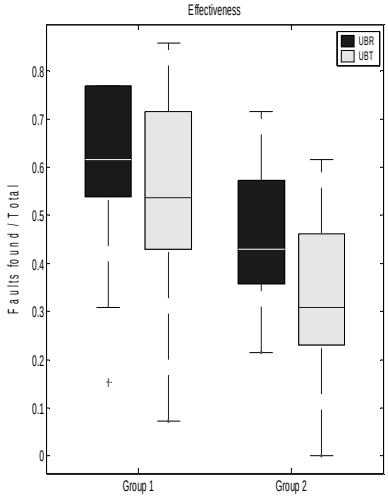

Figure 6. *Effectiveness for group 1 and 2. The black box plots show UBR, while the grey show UBT.*

To investigate the possible influence of the program versions on the results of efficiency and effectiveness, statistical tests comparing version A and version B were conducted. The test on effectiveness does not show any difference between the two versions (p-value = 0.435), confirming that the versions are comparable, with faults that have the same degree of difficulty to be found. The test on efficiency shows higher value for version B (p-value <0.0001), which was expected, as this is the version used the second day. During the second day the subjects were more familiar with the documents and thereby they used less time.

To investigate the influence of the two groups of subjects, the results from Figure 3 and Figure 4 were separated for group 1 and 2. Figure 5 shows box plots of the efficiency of each technique, black boxes show UBR and the grey boxes show UBT, from left to right group 1 and group

Table 4. P-values for the null hypotheses of efficiency and effectiveness. (S) means significant at a 0.05 level.

	Efficiency	**Effectiveness**
Group 1	0.327	0.145
Group 2	<0.001(S)	<0.001(S)

2. The plots for group 1 show no differences in the median value, though with somewhat higher standard deviation for UBT. For group 2 UBR outperforms UBT. Figure 6 shows box plots of the effectiveness of each technique, with similar results, i.e. group 1 has again no major difference in the median value, but still with higher standard deviation for UBT. For group 2 the value for effectiveness is higher for UBR than UBT.

To investigate the significance of the treatments, the hypotheses were tested as described in Section 3.5. For group 1, there was no statistical significance regarding differences between applying UBR or applying UBT, see p-values in Table 4. For group 2, however, statistical significance was obtained, in favour for hypotheses $H_{a\ Eff}$ and $H_{a\ Rate}$, i.e. subjects applying UBR are more efficient and effective than those who apply UBT.

5.3 Faults

The found faults were analyzed in order to evaluate whether there are any differences when using the two techniques. As the number of subjects for each technique was different, we have used the percentage of subjects detecting the faults rather than the number in absolute terms. Figure 7 shows the distribution of subjects applying UBR and subjects applying UBT that have found a certain fault existing in version A. In order to

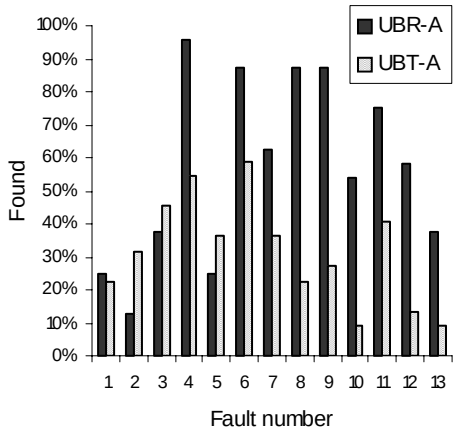

Figure 7. *Percentage of subjects detecting each fault in version A.*

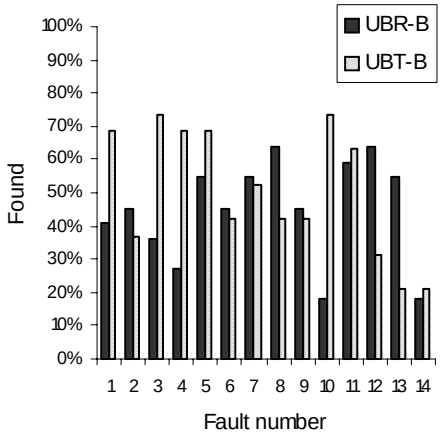

Figure 8. *Percentage of subjects detecting each fault in version B.*

analyze whether the reviewers and testers detected different faults when applying the two techniques (hypothesis H_{Fault}), a chi-square test was used. The result of the chi-square test for version A shows that for the first experiment session, the reviewers and the testers found different faults (p=0.024).

The faults in version B were analyzed by the same method as above. The distribution for the faults in version B that each treatment found is shown in Figure 8. The result of the chi-square test for version B shows, however, not statistical significant difference (p=0.083) at the chosen level of significance.

As mentioned in Section 3.2, five identical faults, in terms of type and position, existed both in version A as well as version B. The distribution for these faults found by each treatment, presented in Figure 9, shows from left to right UBR-A, UBT-B, UBT-A, and UBR-B, i.e. the first two bars show group 1, in the order they applied the techniques, while the third and fourth bars show group 2 in the order they applied the techniques.

A chi-square test was used again to investigate if different faults were found by each technique, though this does not show statistical significant difference between the treatments (p = 0.3927).

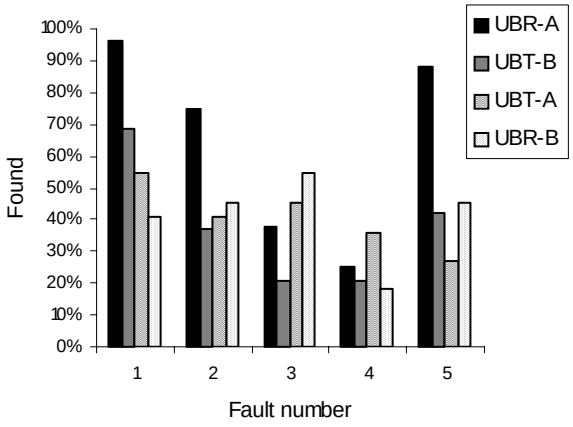

Figure 9. *Percentage of subjects finding the five faults that existed in both versions.*

6. Discussion

In this section we discuss the results of the experiment and the practical implications of the results. The hypotheses of the experiment are summarized as follows:

- H_{Eff} – Usage-based reading is significantly more efficient than usage-based testing, without influence of other factors such as groups and versions of artefacts. However, when separating the data for the two groups, the results for group 1 does not show any significant difference, while the results for group 2 show that usage-based reading is significantly more efficient than usage-based testing.
- H_{Rate} – Usage-based reading is more effective than usage-based testing, without influence of other factors. When separating the two groups, the results for group 1 does not show any significant difference, while the results for group 2 show that usage-based reading is significantly more effective than usage-based testing.
- H_{Fault} – The subjects find different faults when applying the techniques on version A. When applying the techniques on version B the statistical test does not show any statistical difference.

In summary, the inspection technique is better than the testing technique, although there are differences between the two groups. The results for group 1 only, which applied UBR the first day and UBT the second day, show that the two techniques perform similarly. However, the results for group 2, which applied UBT the first day and UBR the second day, show significant differences, in favour of UBR, both regarding efficiency and effectiveness. The results for UBR for the two groups are rather stable, also compared to previous experiments on UBR [22], independently of whether this was the first treatment applied or the second.

The performance for UBT is noticeable improved for group 1, which applied this treatment as the second one, compared to group 2's performance for UBT, which was applied as the first treatment. This is possibly related to that the subjects during the second day were more acquainted with the documents. The differences between the groups can be interpreted as the learning curves for the two techniques are different, i.e. longer for UBT. It is indicated by higher values of efficiency for the

second day, applying the techniques on version B, though it is specifically for UBT that the efficiency values are improved.

The difference between the two techniques cannot be explained by that the two versions of the system are different. The statistical tests show that the fault detection effectiveness is similar for the two versions, while the efficiency is higher for version B. This is expected since the subjects used version B during the second day and were hence more familiar with the documents and thereby used less time, but they still had the same fault rates.

The third hypothesis, whether the techniques find different faults or not, was rejected for version A ($p=0.024$), but not for version B ($p=0.083$). This implies that we interpret this as the techniques are complementary, and both useful for finding different faults, however some faults can as easily be found by either one of the techniques.

The results for the five faults, which were present in both versions have been analyzed to ensure that they have not influenced the results in any way. The statistical tests show no significant difference between the techniques or the groups, and the fault rates for these faults are not higher during the second day. We interpret this, as the faults do not have any resemblance when finding them in the design document during inspection compared to when finding them in the test output traces during testing. We did tell the subjects that there should not exist the same faults in both versions, so they have probably not been searching for them specifically on the second day. The risk that they thereby should not report them is considered very small since they generally reported on a very high number of what they considered as faults. Roughly counting, two third of the reported faults were false positives.

When considering the cost-effectiveness of UBR and UBT, the time used for development of use cases, prioritization of these, development of test cases and execution of these, and rework, are not taken into account in the analysis. The preparation activities were all conducted in advance and the time spent is not included in the analysis. However, we assume the test case generation and execution require more time than the preparation of use cases for inspection, which is in favour of the usage-based reading technique. In particular, since inspection enables earlier fault detection when applied already to the design document, rework costs are reduced, as time is not wasted on first implementing the faulty design and then correcting it later when the faults are found in test.

Previous work in this area has most often been concerned with inspection and testing of code, of which some results are summarized in [13]. The most general pattern is that the techniques find different faults, and hence should be used as complements. Industrial case studies also report benefits from inspection [4], [5]. However, earlier experiments are not focused on design inspection versus functional test as this study.

7. Conclusions

This paper reports an experiment on two fault detection techniques, usage-based reading (UBR) and usage-based testing (UBT), conducted with 51 Master's students in December 2002. The results show that UBR is significantly more effective and efficient than UBT. The results are slightly different for the two experiment groups but the overall results are in favour of UBR. The results indicate that it takes longer to learn the UBT. Comparing the techniques independently of program versions, UBR is significantly better than UBT both in terms of efficiency as effectiveness. The experiment also investigated whether different faults were found by the two techniques. One group shows statistically significant differences while the other does not. If the techniques find different faults, it implies that they should be used as complements.

Given that the results can be replicated and generalized, we conclude that this study provides evidence that usage-based reading for design inspections are more effective and efficient than usage-based functional testing. This holds for the fault detection as such, which is the focus of this paper. When taking the rework costs into account, the potential gains are larger with the inspection technique since it is possible to apply earlier in the development cycle.

Further work should include further experimentation to replicate the results. One focus could be on which types of faults are found by the different techniques. This would give better answers to whether the two techniques find different types of faults or not, and could facilitate the decisions of how to combine inspection and testing, and answer the question about how to reduce these effort consuming activities. Furthermore, conducting an experiment with the testing treatment in a dynamic test environment would reduce the threat to construct validity of having the paper-based approach. However, this would also complicate the treatment and might give rise to an even slower learning curve.

An experiment on an inspection technique and a testing technique is described in this paper, with the main purpose of evaluating and to get knowledge of how to combine these verification and validation techniques. One experiment is not enough to answer this question. However, where no other experimental evidence is available, our results may represent a data point, which can be used to direct future work in this area.

Acknowledgement. This work was partly funded by The Swedish National Agency for Innovation Systems (VINNOVA), under a grant for the Center for Applied Software Research at Lund University (LUCAS). We thank the students that participated in the experiment.

8. References

[1] Aurum, A., Petersson, H. and Wohlin, C., "State-of-the-Art: Software Inspections after 25 Years", *Software Testing, Verification and Reliability*, 12(3):133-154, 2002.

[2] Basili, V. R. and Selby, R. W., "Comparing the Effectiveness of Software Testing Strategies", *IEEE Transaction on Software Engineering*, 13(12):1278-1296, 1987.

[3] Basili, V. R., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sørumgård, S. and Zelkowitz, M. V., "The Empirical Investigation of Perspective-Based Reading", *Empirical Software Engineering: An International Journal*, 1(2):133-164, 1996.

[4] Berling, T. and Thelin, T., "An Industrial Case Study of the Verification and Validation Activities", *Proceedings of the 9th International Symposium on Software Metrics*, 2003.

[5] Conradi, R., Marjara, A. S., and Skåtevik, B., "Empirical Study of Inspection and Testing Data", *Proceedings of the 1st International Conference on Product Focused Software Process Improvement*, pp. 263-284, 1999.

[6] Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development", *IBM Systems Journal,* 15(3):182-211, 1976.

[7] Hetzel, B., *The Complete Guide to Software Testing*, John Wiley & Sons, 1988.

[8] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990, Corrected Edition February, 1991.

[9] ITU-T Z.120 *Message Sequence Charts*, MSC, ITU-T Recommendation Z.120, 1996.

[10] Juristo, N. and Moreno, A. M., *Basics of Software Engineering Experimentation*, Kluwer Academic Publisher, 2001.

[11] Juristo N., Moreno A. M. and Vegas S., "A Survey on Testing Technique Empirical Studies: How Limited is Our Knowledge", *Proceedings of the 1st International Symposium on Empirical Software Engineering*, pp. 161-172, 2002.

[12] Kamsties, E. and Lott, C. M., "An Empirical Evaluation of Three Defect-Detection Techniques", *Proceedings of the 5th European Software Engineering Conference*, pp. 362-383, 1995.

[13] Laitenberger, O., "Studying the Effects of Code Inspection and Structural Testing on Software Quality", *Proceedings of 9th International Symposium on Software Reliability Engineering*, pp. 237-246, 1998.

[14] Musa, J. D., "Operational profiles in software-reliability engineering", *IEEE Software*, 10(2):14-32, 1993.

[15] Reid, S. C., "An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing", *Proceedings of the 4th International Software Metrics Symposium*, pp. 64-73, 1997.

[16] Roper, M., Wood, M. and Miller, J., "An Empirical Evaluation of Defect Detection Techniques", *Information and Software Technology*, 39(11):763-775, 1997.

[17] Runeson, P. and Regnell, B., "Derivation of an Integrated Operational Profile and Use Case Model", *Proceedings of the 9th International Symposium on Software Reliability Engineering*, pp. 70-79, 1998.

[18] Runeson, P., "Using Students as Experiment Subjects – An Analysis on Graduate and Freshmen Student Data", *Proceedings of the 7th International Conference on Empirical Assessment & Evaluation in Software Engineering*, pp. 95-102, 2003.

[19] Siegel, S. and Castellan, N. J., *Nonparametric Statistics for the Behavioral Sciences*, McGraw-Hill, 1988.

[20] So, S. S., Cha, S. D., Shimeall, T. J. and Kwon, Y. R., "An Empirical Evaluation of Six Methods to Detect Faults in Software", *Software Testing, Verification and Reliability*, 12(3):155-172, 2002.

[21] Thelin, T., Runeson, P. and Regnell, B., "Usage-Based Reading – An Experiment to Guide Reviewers with Use Cases", *Information and Software Technology*, 43(15):925-938, 2001.

[22] Thelin, T., Runeson, P., Wohlin, C., Olsson, T. and Andersson, C., "How Much Information is Needed for Usage-Based Reading? – A series of Experiments", *Proceedings of the 1st International Symposium on Empirical Software Engineering*, pp. 127-138, 2002.

[23] Thelin, T., Runeson, P. and Wohlin, C., "An Experimental Comparison of Usage-Based and Checklist-Based Reading", *IEEE Transactions on Software Engineering*, 29(8):687-704, 2003.

[24] Thelin, T., Runeson, P. and Wohlin C., "Prioritized Use Cases as a Vehicle for Software Inspections", *IEEE Software*, 20(4):30-33, 2003.

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

PAPER III

Adaptation of a Simulation Model Template for Testing to an Industrial Project

Tomas Berling, Carina Andersson, Martin Höst, Christian Nyberg

Proceedings of 2003 Software Process Simulation Modeling Workshop, 2003.

III

Abstract

Process understanding and improvements are essential in software industry in order to achieve cost effectiveness and short delivery times. One means of increasing process understanding and improvement is to utilize software process simulation.

This paper describes how a template model was created in order to increase the knowledge of the code development and test processes for an industrial organization. The template model was created from an existing system dynamics model for the unit test phase. The paper shows how the template model can be adapted and extended to fit a similar organization. The simulation model is applied for investigating the relationship between defect prevention in the development phase and defect detection in the various test phases. Data from a large contract-driven project were used in a case study to calibrate the adapted and extended model, which included code development and four test phases. Programmers and testers were involved in the design of the model.

The results show that it is possible to use the introduced template model and to adapt and extend it to a specific organization. We can also conclude that it is important to involve project members who contribute to the model building. The process understanding of the participating project members is increased due to their involvement.

1. Introduction

Simulation involves experimentation with a model of a system instead of the system itself. Usually the model of the system is implemented in a computer program. Some reasons for the increasing interest of using simulations in industry are:

- It might be dangerous to experiment with the system. If for example the system is a nuclear power plant, experimentation with a new control system is not allowed until it is simulated.
- The system might not exist. If for example a new aircraft is constructed, it is best to evaluate its performance using simulation before actually building it. It would be too expensive to build several different aircraft and measure their performance.
- Before changing an organization it is advisable to simulate the new organization to see if it meets the demands put on it.

The models used in simulation usually consist of a state description and a number of rules that describe how the state is changed with time, given a certain environment. The rules of change can be differential or difference equations.

Usually a distinction is made between discrete event simulation and continuous simulation [11]. In discrete event simulation the state of a system is changed only when certain events occur and is not changed between these events. A typical example is a queuing system where the state is the number of customers in the queue and the events are arrivals of customers and departures of customers. An example of a continuous simulation is when the air pressure around an aircraft is simulated as a function of time. Usually differential equations are used to describe state changes in models used for continuous simulation. It is also possible to combine discrete event simulation and continuous simulation, which is usually called hybrid simulation, see for example Donzelli et al. [7], and Martin et al. [13].

In software engineering the main reasons for using simulations of software processes are for the purpose of strategic management, planning, control and operational management, process improvement and technology adoption, understanding, and training and learning [1], [10]. In a software development project the effect of a process change in the code development or the test phases can be difficult to predict or it can be

difficult to prioritize work in the different phases during time pressure, for example. A simulation model is appropriate to use in these cases. The risk of changing processes in the running projects in order to learn about it and to implement new ideas is too high, since it would lead to longer delivery times and high costs. A simulation model is used without any risk and with a relatively low cost.

The focus of this study is to enhance the modelling of the code development and test phases, for any organization, in order to understand the current software development process and to facilitate for future improvements to these processes. A system dynamics model with a code development phase and a test phase has been developed, which can be used as a template for other organizations to simulate these phases. The paper describes how this template model can be extended and adapted to suite the software development process in an organization.

The template model has been extended and adapted at Ericsson Microwave Systems AB, Sweden, to facilitate process improvements. Specifically the resources used, the distribution of undiscovered defects in the different test phases, and the cost of finding defects in different phases were studied.

The main research questions of this study are:

- What key tasks, primary objects, and vital resources, in the simplest case, are needed in a simulation model in order to investigate for example the resources used, the distribution of undiscovered defects in different phases, and the cost of finding defects in different phases?
- How can such a template model be adapted and extended to a specific organization?

The template model in this study is based on the study by Collofello et al. [6], who modelled and simulated a unit test phase. The idea of viewing the unit test phase as two flows, a testing flow and a detection flow originates from Collofello et al., and in this study the model is further generalized.

Modelling and simulation of the code development and test phases have been performed in other studies. Analysis of the test process has for example been performed by Raffo et al. [14] in which the impact of a process change was simulated. The process change involved the implementation of unit test plans and the simulation result showed that the process change would be successful. Madachy et al. [12] have

simulated the peer review model in an organization to investigate the dynamic project effects of performing inspections. The code development and test phases are parts of this model. The simulation results helped the planning and performance of peer reviews. Andersson et al. [2] simulated the requirements specification and test phases and specifically analysed the resource allocation in the different activities to decrease the project cycle time. The models used in these studies are specific for the examined organizations in contrast to the general model presented here.

In this paper a continuous simulation model is used. A discrete event simulation can also be used for this purpose. The discrete event simulation technique has for example been used to model a specific requirements management process for identification of overload situations [8].

The paper is structured as follows: The organization, developed products, and process are described in the environment part in Section 2. The method used is presented in Section 3 and the model and simulation is reported in Section 4. Conclusions are presented in Section 5.

2. Environment

2.1 Organization and developed products

The study is performed at Ericsson Microwave Systems AB, where radar systems are developed. The systems are large and complex with hard real-time constraints. The systems are divided into sub-systems, which are integrated at several levels, both hardware and software wise.

The products are delivered on contract. There are therefore relatively few customers compared to broad market products.

2.2 Process

The organization follows an incremental software development process. In each development step, called increment, functionality is added to the previous one. The functionality is added in a manner so that the system is always executable. The first increment contains only basic functionality and the last increment contains all functions.

In each increment the following development phases are included:

- System requirements specification
- Sub-system level 1 requirements specification (see Figure 1 for the different sub-system levels)
- Sub-system level 2 requirements specification
- Code development and unit test
- Sub-system level 2 verification
- Sub-system level 1 verification
- System integration
- System verification

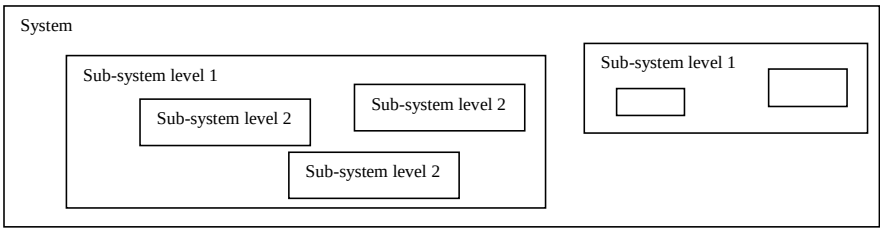

Figure 1. *The sub-system level 2 in the study in relation to the whole system.*

System acceptance tests with the customer are performed after the last increment. Table 1 presents the development phases included in the case study and the personnel performing it.

The sub-system level 1 requirements specification phase is performed by design engineers and the sub-system level 2 requirements specification phase is performed by programmers. These two phases are not included in the simulation study.

The sub-system is developed by approximately 4 programmers on average. The sub-system is divided into units, which are tested separately. The unit tests are developed and executed at the same time as the code development for the system. When the programmers have completed the code development and the unit tests are executed without failures the code is frozen in a unique revision and the next phase, sub-system level 2 verification, is performed. In sub-system level 1 verification, which is the next phase, the sub-systems at level 2 are integrated and verified into one sub-system at level 1. When this phase is completed the sub-system at

Table 1. Development phases in the study and the personnel performing it.

Development phase	**Personnel**
Code development and unit test	Programmers
Sub-system level 2 verification	Programmers
Sub-system level 1 verification	Programmers
System integration	Independent testers
System verification	Independent testers

level 1 is delivered to the independent test engineers. In the system integration phase the testers integrate the sub-system level 1 with several other sub-systems at level 1. When the integration phase is conducted the next phase, system verification, is performed. In the system verification phase the system is verified by the testers. When the system has been verified and defects have been corrected or postponed, the development of the increment has been completed.

Several increments can exist at the same time, but in different phases, i.e. the next increment can start before the previous is completed. Figure 2 shows an example of development phases and increments in an incremental development process. Figure 1 shows the sub-system level 2 in the study in relation to the whole system.

The organization also follows a formal review process for all documents. All necessary documents are defined in the formal incremental software development process.

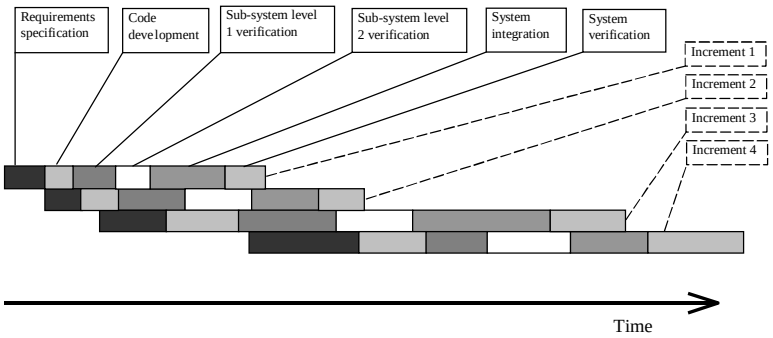

Figure 2. *An example of development phases and increments in an incremental development process.*

3. Method

In order to answer the research questions, the idea of implementing a template model, and adapting and extending it to a specific organization is examined. Building the simulation model was an iterative procedure with a continuous contact with the programmers and testers in the modelled project. The close co-operation with the programmers and testers resulted in discussions on both model purpose, and model structure. The development procedure can be described in several steps, where feedback from the programmers and testers was received in every step.

The first step concerned specifying the purpose, model scope, result variables, process abstraction, and input parameters. This was performed according to a guideline of Kellner et al. [10]. These aspects were identified in order to specify what to simulate.

The *purpose* of the simulation study is to enhance the understanding of the code development and testing phases, specifically the resources used, the distribution of undiscovered defects in the different test phases, and the cost of finding defects in different phases. When the understanding has increased the simulation model can be used for process improvement and technology adoption in the code development and test phases.

The *model scope* was confined to the development and testing phases. The requirements specification phases were excluded from the model's boundary for the reason that faults in the requirements specifications are only indirectly causing defects in the code, through the programmers' knowledge and skills. If the code would have been generated automatically from the requirements specifications the requirements specification phases would have been included. Even though the requirements specification phases are not unique parts of the model, they could be included as input parameters at the development, and testing phases.

The simulation of an industrial project was performed for one increment, see Figure 2, i.e. a portion of a life cycle, in one project.

The *result variables* in the project simulation included defect distribution between the phases, resources used in the phases, and an estimated cost of finding defects in different phases.

The *process abstraction* part is the key contribution of this paper. The main research questions presented in Section 1 yield the process abstraction. The key tasks, primary objects, and vital resources according

to Table 2 were identified as the simplest case for the template model. The idea of viewing the unit test phase as two flows, a testing flow and a detection flow originates from Collofello et al. [6]. The template model is built from this idea.

This template model can be adapted and extended with further key tasks, primary objects, and vital resources to suite the industrial environment.

Table 2. Key tasks, primary objects, and vital resources for the template model.

Key tasks	Primary objects	Vital resources
Code development	Incoming work in KLOC	Programmers
Testing of code	Defects in code	Testers

The key tasks, primary objects, and vital resources in the industrial simulation included in this study were adapted and extended according to Table 3. This adaptation and extension is directly related to the organization process, described in Section 2.2.

The key tasks are the activities relevant to the model purpose, while the primary objects are the project artefacts, believed to affect the result variables. Vital resources could also be the hardware used for code development and testing, but this was not included in this industrial simulation.

A case study by Berling and Thelin [4] of the verification and validation activities in the organization served as a baseline for the important factors and the expected behaviour of the simulated system. In their study, the trade-off between inspection and testing, in terms of faults found and resources used were investigated in the organization. Data

Table 3. Key tasks, primary objects, and vital resources for the industrial environment in this study.

Key tasks	Primary objects	Vital resources
Code development and unit test	Incoming work in KLOC	Programmers
Sub-system level 2 verification	Defects in code	Testers
Sub-system level 1 verification		
System integration		
System verification		
Rework of defects, i.e. corrections		

from their study were used to calibrate and validate the adapted simulation model.

The input parameters are defined in accordance with the desired result variables and the process abstraction. In the template model the in-parameters were defined according to Table 4.

In the industrial simulation included in this study the template model was extended with further in-parameters. Most of the in-parameters in the industrial setting are constants, defined by the model user before the simulation model is executed, while others are varying over time. The input parameters are described in more detail in Appendix A.

With the simulation purpose, the model scope, the result variables, the key tasks, and the input parameters in mind the template model was adapted and extended to a first draft on paper. The draft model only consisted of qualitatively affecting relationships, and was without weighting and quantitative relationships.

To ensure the validity of the draft model, feedback was received from programmers and testers on the included in-parameters and relationships. Walkthroughs of the model were performed. Their comments mainly concerned definitions and effects of in-parameters and cost aspects of finding defects in different phases. Test coverage was for example one in-parameter added to the model after comments from the programmers and testers.

According to the programmers' and testers' comments the model was revised and thereafter transformed into the simulation tool. A visual description was chosen in order to enhance the understanding of the model, and to ease the calibration of the model, which continuously was performed with assistance from programmers and testers.

The development of the model extended from the template model, with few affecting factors, to a more detailed and project specific model

Table 4. The in-parameters in the template model.

Input parameters
Incoming work
Programmer resource
Tester resource
Coding method
Testing method

with more relationships and inter-dependencies. As a result of the study by Berling and Thelin [4] the factor "Low-level design" was added to the model. This factor became apparent when faults found in the real system were classified and analysed, i.e. faults were injected in the real system due to an inadequate low-level design for the sub-system.

In addition to the walkthroughs with programmers and testers, and using their estimates based on their past experiences, calibration of the simulation model was performed with real project data, see Section 4.2 for a description. If project data are not available statistical data from literature can be used initially, see for example Jones [9].

The opportunity to further develop the model still exists, either to include or exclude activities, if these are assumed to affect the output, or to make changes to adapt the model for another development project.

4. Model and simulation

4.1 Template model

The template model, including only the necessary key objects in the simplest case, is presented in Figure 3. This model consists of one module for code development, module A, and one module for test, module B. The arrow in Figure 3 corresponds to undiscovered defects, which are transferred from module A to module B. With the template model the user can simulate the number of injected defects during code development and the number of detected defects during testing as well as used resources and the time for development and testing. When the template model behaviour is understood by the user, the model can be extended and adapted to reflect the industrial setting. This is described in the next section.

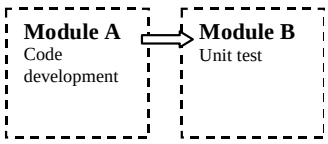

Figure 3. *The template model with one development phase and one test phase*

The code development module, module A, is modelled according to Figure 4. The model user estimates the following input parameters:

- The incoming work
- The number of programmers
- The average number of injected defects per day per programmer
- The average produced number of KLOC per day per programmer with the coding method

The values of the input parameters can be estimated by project measures, reported statistics, or best estimates from experienced programmers and testers. The lower flow in Figure 4 corresponds to the coding rate, which is determined by the number of programmers and the average KLOC per day produced per programmer. The number of KLOC in incoming work together with the coding rate determine the number of days it takes to complete the code. The upper flow in Figure 4 corresponds to the defect injection rate during coding, which is determined by the coding rate and the injected number of defects per KLOC by the programmers, due to the coding method. The output from the module is the number of undiscovered defects in the code, which is transferred to undiscovered defects in module B, unit test, when module A has been completed.

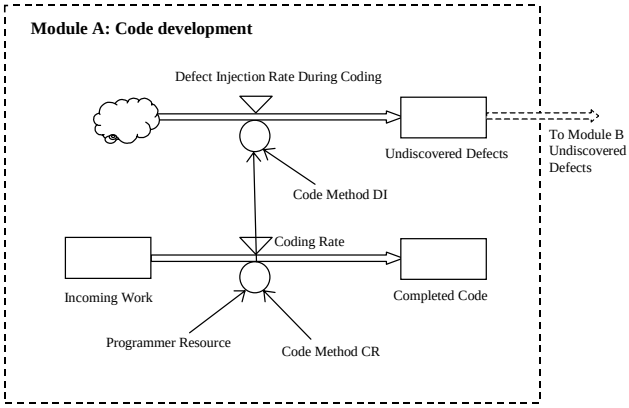

Figure 4. *The development module, module A, in the template model.*

The formulas used in module A are:

$$CodingRate = CodingMethodCR \times ProgrammerResource$$

$$DefectInjectionRateDuringCoding = CodingRate \times CodingMethodDI$$

The Coding Method is divided into Coding Method CR for the coding rate, in which the unit is KLOC/day, and Coding Method DI for the defect injection rate, in which the unit is the number of defects/KLOC. The input parameters in module A and their units are listed in Table 5.

Table 5. Input parameters in module A.

Input parameter	**Unit**
Incoming work	KLOC
Programmer resource	Number of programmers
Coding Method CR	KLOC/day
Coding Method DI	Number of defects/KLOC

The test module, module B, is modelled according to Figure 5. The incoming work, the number of testers, the average number of detected defects per day per tester, and the average number of KLOC tested per day per tester with the test method is estimated by the model user. The lower flow in Figure 4 corresponds to the testing rate, which is determined by the number of testers and the average KLOC tested per day per tester. The number of KLOC in incoming work together with the testing rate determine the number of days it takes to test the code. The upper flow in Figure 5 corresponds to the defect detection rate, which is determined by the testing rate and the detected number of defects per KLOC with the test method. The output from the module is the number of undiscovered defects in the code. The formulas used in module B are:

$$TestRate = TestMethodTR \times TesterResource$$

$$DefectDetectionRate = TestRate \times TestMethodDD$$

The Test Method is divided into Test Method TR for the testing rate, in which the unit is KLOC/day, and Test Method DD for the defect detection rate, in which the unit is the number of defects/KLOC. The

detection rate is independent of the number of faults in the code. This was chosen for practical reasons.

The input parameters in module B and the units are listed in Table 6. The number of tested KLOC per day is more difficult to estimate than for example the number of tested requirements per day. The unit number of tested KLOC per day is used anyway in order for the test method to be estimated in number of defects per KLOC in the upper flow. The unit in the upper flow would otherwise be the number of defects per requirement, which is also a difficult unit. A suggestion for the model user is to approximate that each requirement is of equal size in KLOC. A model extension with the input parameter test coverage, for example, can be performed by measuring test coverage by the number of requirements tested, and then approximating the corresponding number of KLOC tested. This approximation is used in this industrial simulation.

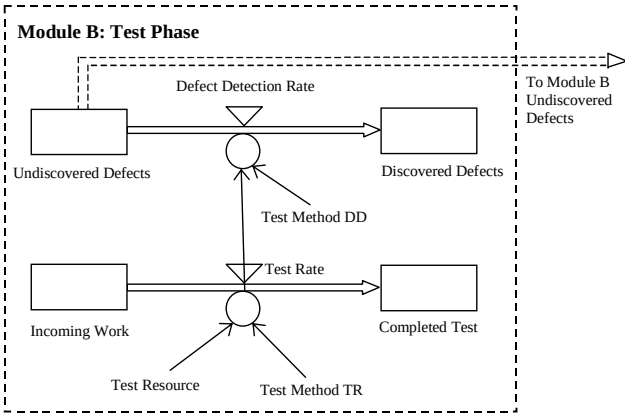

Figure 5. *The test module, module B, in the template model.*

Table 6. Input parameters in module B.

Input parameter	Unit
Incoming work	KLOC
Test resource	Number of testers
Test Method TR	KLOC/day
Test Method DD	Number of defects/KLOC

4.2 Simulation with an adapted model in an industrial setting

The template module described in Section 4.1 was implemented, extended and adapted in the organization described in Section 2. The in-parameters listed in Table A1 in Appendix A were considered important for module A. The major adaptations in module A are the inclusion of unit tests in the development phase and the extension of in-parameters to the code rate and defect injection rate, see Figure 6. The unit test is included in module A, since it is developed and executed in parallel with the code development in the same phase, see Section 2. The in-parameters listed in Table A2 in appendix A were considered important for module B.

The major adaptations in module B from the template model are the extension of a rework flow and the extension of in-parameters to the test rate and defect detection rate, see Figure 7. The rework flow was added in order to estimate resources and time for the corrections of defects. Module B was also extended with a defect injection flow, due to the fact that new defects could be injected in the system during defect correction, see the flow from the cloud in the upper part of Figure 7. The arrow from Defect Rework Rate to Injected Defects due to Rework is added in order to control the number of new injected defects, which is dependent on the

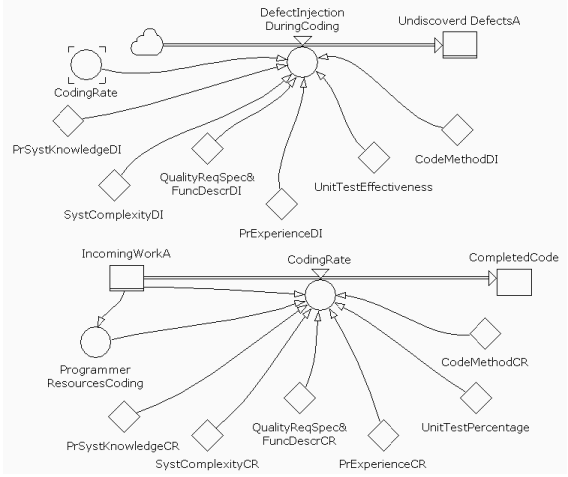

Figure 6. *Module A in the extended and adapted model in the industrial setting.*

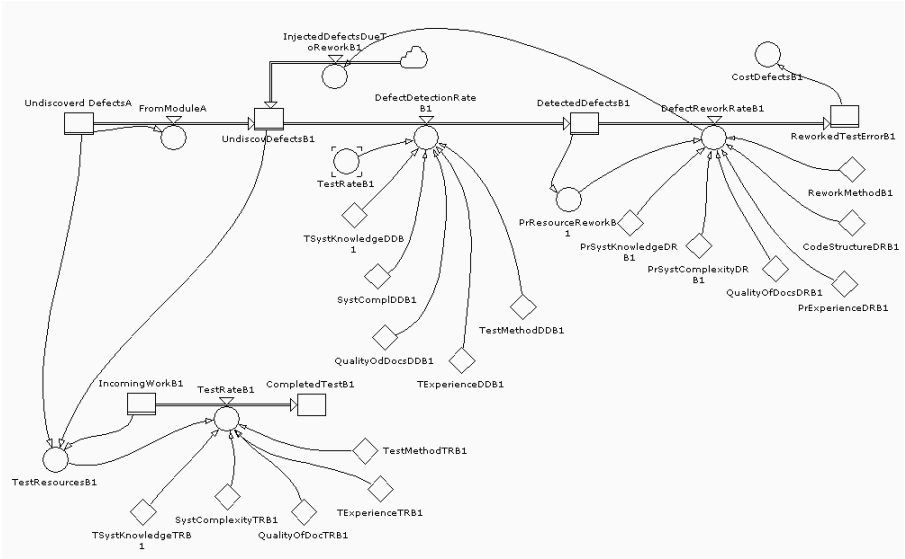

Figure 7. *Module B in the extended and adapted model in an industrial setting.*

number of corrected defects. The flow from Undiscovered Defects A to Undiscovered Defects B1 is added in order to transfer the Undiscovered Defects from module A to module B when module A is completed. The arrows from Undiscovered Defects A, Undiscovered Defects B1, and Incoming Work B1 to Test Resources B1 are added in order to start module B when the Undiscovered Defects have been transferred from module A to module B. The arrow from Detected Defects B1 to PrResource Rework controls that programmer resources are only correcting defects if defects are discovered. The test coverage is controlled by multiplying the Incoming Work with the percentage of test coverage in module B.

The model was also extended with a modified module B for each testing phase, according to the software development process described in Section 2. The model for the industrial setting includes four modules of type B, according to Figure 8. The modules of type B are identical, but the parameters' values differ between the modules to reflect the situation in each phase. Defects from other sub-systems at level 1 and 2 are not included in this study.

The model in-parameters were adjusted to correspond to a real increment in a project with the programmers' and testers' viewpoint on the magnitude of the parameters. The data, which were used to calibrate

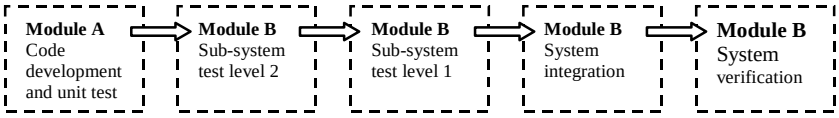

Figure 8. *Adapted and extended model in an industrial setting.*

the model, were taken from a problem reporting system, a personnel time logging system, and the number of lines of code from project data, as well as experiences from the programmers and testers. For more details on these data see [4]. The results of the simulation are presented in Figure 9.

The upper graph in Figure 9 shows the number of undiscovered defects in the different phases. The first steadily growing curve corresponds to the defect injection during coding and unit test. The decrease of the number of undiscovered defects in the test phase curves corresponds to the discovered defects during the test phases. The low increase of undiscovered defects in the test phase curves (see third description in upper part of Figure 9) is due to a low injection rate of new defects in the project during defect corrections. The verification phase in this increment was not performed. The undiscovered defects are therefore not reduced in the last curve, in this case. The model was calibrated to correspond to the real time scale and to the number of defects in the increment. The programmers and testers adjusted the in-parameters to simulate a process change to see how the model worked. The simulation results reflected the simulated change in the number of detected defects in the different phases.

The middle graph shows the number of persons, i.e. resources used in the different phases. The presented resources also include the programmers doing rework during defect corrections. The times in which the resources are zero are due to the model implementation. In this model the transfer of undiscovered defects from one module to another is completed before the next module testing is initiated. The model can be further developed in this respect.

The lower graph shows an estimate of the cost of detecting defects in the various phases. The cost of finding and correcting a defect, in the adapted model, is modelled to be increased for each test phase. This corresponds to the increased cost of performing all test phases again for the corrected defect. The actual flow or performance of new corrected releases of code, due to defect corrections, is not simulated in the model. The largest cost curve in the lower graph is due to the undiscovered

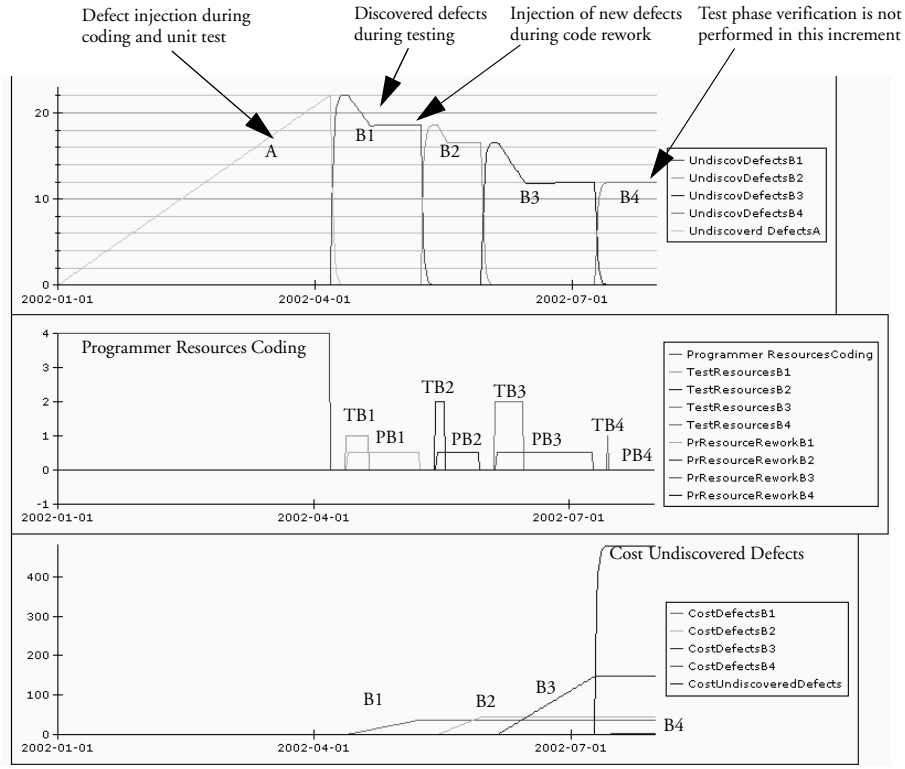

Figure 9. *Results from the simulation model in the industrial setting.*

defects, which were not found in this increment. The cost of finding defects in different phases is difficult to estimate, since each defect can cause different costs. Various approximations and definitions can be used. In this case the cost of finding faults in different phases were approximated with a fictitious value of 10 for the first testing phase, 20 for the next, and so on. The total cost, which is not shown in Figure 9, of all the found and not found defects in the different phases yields a good estimate for a process change in terms of costs.

4.3 Validation

The adapted model was validated with a sensitivity analysis [3]. In the sensitivity analysis the output variables Number of faults, and Calendar time were measured for module A when changing the input parameters Code Method Code Rate, Programmer Resource, Unit Test Percentage, Code Method Defect Injection, and Incoming work at “extreme values”.

The parameter Code Method Code Rate includes the parameters PrSystKnowledge CR, SystComplexity CR, QualityReqSpec&FuncDescr CR, PrExperience CR, and CodeMethod CR. The parameter Code Method Defect Injection includes the parameters PrSystKnowledge DI, SystComplexity DI, QualityReqSpec&FuncDescr DI, PrExperience DI, and CodeMethod DI. This simplification can be performed since these parameters technically are summarized into one parameter in the model.

The "extreme values" of the parameters were chosen by selecting a reasonably high and low value in a range for which the model is used. The number of programmers was for example 2 in the lower limit and 8 in the upper limit. The sensitivity analysis was performed with a full factorial design [5] for module A, which results in 32 runs (5 parameters with 2 levels). The factorial design analysis showed that the number of injected faults are dependent on, and only on, the parameters Code Method Defect Injection, and Incoming work, and in fact Code Method Defect Injection*Incoming work. This is a correct behaviour of the simulated system. The validation of the Calendar time showed that the Unit Test Percentage had been incorrectly implemented, since the calendar time increased when the unit test was reduced. The model was corrected and a validation was performed a second time with a correct behaviour for all parameters.

The validation of module B was performed with the parameters Test Method Test Rate, Test resource, Test Method Defect Detection, Programmer Resource Rework, Incoming work, Rework Method, and Injected Defects due to rework. A simplification of the parameters Test Method Test Rate, Test Method Defect Detection, and Rework Method was performed, similarly as for module A. The output variables were the number of detected faults, the number of days for rework, and the number of test days. These output variables are used for the calculation of costs etc. A fractional factorial design with 16 runs was performed. This means that first-order effects cannot be separated from third-order interactions, but effects of third-order interactions are not considered likely in this case, thus not affecting the result. When choosing the "extreme values" for module B, certain relationships between parameters set limitations. For example the test rate (KLOC/day) could not be greater than the incoming work (KLOC) if the time step is set to 1 day. The analysis of the fractional factorial design showed that the model behaved correctly for all parameters and output variables.

5. Conclusions

In this study a template model has been developed and evaluated. The template model has been specialised into a model that is adapted to a specific industrial project. We have found that it is possible to use the template model when a specific model is derived, and that it is possible to derive the specialised model as it was done in the presented case study. We have also seen that it is important to involve representatives from the project. In the case that is presented, the representatives came from the project that was simulated, and we believe that this is a feasible way in cases where this is possible. The programmers and the testers had many important suggestions and corrections in the work with the specific model.

It is also concluded that a thorough analysis of project data, yielding information regarding resources used, faults found etc. in the phases facilitate the model building and validation.

During the feedback-session it was found that the programmers and testers were interested and they thought that they had gained understanding of the process because of this work. We therefore believe that the model describes issues that are important, and that it is a good representation of the real process.

We believe that it is possible to use the template model in organisations that are similar to the studied organisation. It is probably possible to adapt the model in the same way as in this study, if the project does not differ very much.

Further work includes more experimentation with the template model. For example, the organisation in the case study is planning to use the adapted and extended model in more increments.

Acknowledgement. The authors would like to thank our colleagues at Ericsson Microwave Systems AB Maria Jonsson, Reine Larsson, Magnus Larsson, Carl-Ejnar Bergh, and Thomas Svensson for their contribution to this work.

This work was partly funded by The Swedish Agency for Innovation Systems (VINNOVA), under a grant for the Centre for Applied Software Research at Lund University (LUCAS).

6. References

[1] Abdel-Hamid, T. and Madnick, S., *Software Project Dynamics: An Integrated Approach*, Englewood Cliffs, New Jersey, Prentice Hall, 1991.

[2] Andersson, C., Karlsson, L., Nedstam, J., Höst, M. and Nilsson, B. I., "Understanding Software Processes through System Dynamics Simulation: A Case Study", *Proceedings of 9th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pp. 41-48, 2002.

[3] Banks, J., *Handbook of Simulation*, John Wiley & Sons, 1998.

[4] Berling, T. and Thelin, T. "An Industrial Case Study of the Verification and Validation Activities", *Proceedings of 9th International Software Metrics Symposium*, pp. 226-238, 2003.

[5] Box, G. E. P., Hunter, W. G., and Hunter, J. S., S*tatistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*, Wiley-Interscience, 1978.

[6] Collofello, J. S., Zhen Yang, Tvedt, J. D., Merrill, D. and Rus, I., "Modeling Software Testing Processes", *Proceedings of the 15th IEEE International Phoenix Conference on Computers and Communications,* pp. 289-293, 1996.

[7] Donzelli, P. and Iazeolla, G., "Hybrid Simulation Modelling of the Software Process", *Journal of Systems and Software*, 59(3):227-235, 2001.

[8] Höst, M., Regnell, B., Natt och Dag, J., Nedstam, J., and Nyberg, C. "Exploring Bottlenecks in Market-driven Requirements Management Processes with Discrete Event Simulation", *Journal of Systems and Software*, 59(3):323-332, 2001.

[9] Jones, T. C., *Estimating Software Cost*, McGraw-Hill, 1998.

[10] Kellner, M. I., Madachy, R. J. and Raffo, D. M., "Software Process Simulation Modeling: Why? What? How?", *Journal of Systems and Software*, 46(2-3):91-105, 1999.

[11] Law, A. M, and Kelton, W. D., *Simulation Modeling and Analysis* (3rd ed.), McGraw-Hill, 2000.

[12] Madachy, R. and Tarbet, D. "Case Studies in Software Process Modeling with System Dynamics", *Software Process Improvement and Practice*, 5(2-3):133-146, 2000.

[13] Martin, R. and Raffo, D. "Application of a Hybrid Process Simulation Model to a Software Development Project", *Journal of Systems and Software*, 59(3):237-246, 2001.

[14] Raffo, D. M. and Kellner, M. I., "Analyzing Process Improvements Using the Process Tradeoff Analysis Method", *Proceedings of the Software Process Simulation Modeling Workshop*, 2000.

Appendix A

Table A1. Important in-parameters for module A in the extended and adapted model.

Input parameters module A	Measure
Programmers' System Knowledge Code Rate	Consider the characteristics in Table A3 below "Programmer participation in reviews", "Number of years with total system", "Number of years with sub-system", and "Used system in laboratory environment". Estimate a measure on the reduced or increased production of KLOC/day, due to level of system knowledge.
System Complexity Code Rate	Consider the characteristics in Table A3 below "Common components", "Sub-system's control", and "Other sub-systems' control". Estimate a measure on the reduced or increased production of KLOC/day, due to level of system complexity.
Quality of Requirements Specifications and Functional Descriptions Code Rate	Consider the characteristics in Table A3 below "Documentation status", "Review of documents", and "Faults in documents". Estimate a measure on the reduced or increased production of KLOC/day, due to level of quality of requirements specifications and functional descriptions.
Programmers' Experience Code Rate	The programmers' experience as a programmer and with the language used. Estimate a measure on the reduced or increased production of KLOC/day, due to level of programmers' experience.
Coding Method Code Rate	The number of produced KLOC/day. Estimate a measure of the number of produced KLOC/day per programmer, due to the coding method used.
Programmer Resource for Coding	The number of programmers for the sub-system development.
Amount of Incoming Work (KLOC)	Lines of uncommented code.
Programmers' System Knowledge Defect Injection	Consider the characteristics in Table A3 below "Programmer participation in reviews", "Number of years with total system", "Number of years with sub-system", and "Used system in laboratory environment". Estimate a measure on the number of injected defects/ KLOC, due to level of system knowledge.

Table A1. Important in-parameters for module A in the extended and adapted model.

Input parameters module A	**Measure**
System Complexity Defect Injection	Consider the characteristics in Table A3 below "Common components", "Sub-system's control", and "Other sub-systems' control". Estimate a measure on the number of injected defects/KLOC, due to level of system complexity.
Quality of Requirements Specifications and Functional Descriptions Defect Injection	Consider the characteristics in Table A3 below "Documentation status", "Review of documents", and "Faults in documents". Estimate a measure on the number of injected defects/KLOC, due to level of quality of requirements specifications and functional descriptions.
Programmers' Experience Defect Injection	The programmers' experience as a programmer and with the language used. Estimate a measure on the number of injected defects/KLOC, due to level of programmers' experience.
Unit Test Effectiveness	The number of defects/KLOC discovered by unit test.
Coding Method Defect Injection Rate	The number of injected defects/KLOC. Estimate a measure of the number of injected defects/KLOC per programmer, due to the coding method used.

Table A2. Important in-parameters for module B in the extended and adapted model.

Input parameters module B	**Measure**
Testers' System Knowledge Test Rate	Consider the characteristics in Table A3 below "Tester participation in reviews", "Number of years with total system", "Number of years with sub-system", and "Used system in laboratory environment". Estimate a measure on the number of tested KLOC/day, due to level of system knowledge.
System Complexity Test Rate	Consider the characteristics in Table A3 below "Common components", "Sub-system's control", and "Other sub-systems' control". Estimate a measure on the number of tested KLOC/day, due to level of system complexity.

Table A2. Important in-parameters for module B in the extended and adapted model.

Input parameters module B	Measure
Quality of Requirements Specifications and Functional Descriptions Test Rate	Consider the characteristics in Table A3 below "Documentation status", "Review of documents", and "Faults in documents". Estimate a measure on the number of tested KLOC/day, due to level of quality of requirements specifications and functional descriptions.
Testers' Experience Test Rate	The testers' experience. Estimate a measure on the number of tested KLOC/day, due to level of testers' experience.
Test Method Test Rate	The number of tested KLOC/day. Estimate a measure of the number of tested KLOC/day per tester, due to the test method used.
Test Resource	The number of testers in the test phase.
Incoming Work	The number of KLOC of the sub-system to be tested.
Test Coverage	The % of code tested. (Multiplied with incoming work in the model)
Testers' System Knowledge Defect Detection	Consider the characteristics in Table A3 below "Tester participation in reviews", "Number of years with total system", "Number of years with sub-system", and "Used system in laboratory environment". Estimate a measure on the number of detected defects/KLOC, due to level of system knowledge.
System Complexity Defect Detection	Consider the characteristics in Table A3 below "Common components", "Sub-system's control", and "Other sub-systems' control". Estimate a measure on the number of detected defects/KLOC, due to level of system complexity.
Quality of Requirements Specifications and Functional Descriptions Defect Detection	Consider the characteristics in Table A3 below "Documentation status", "Review of documents", and "Faults in documents". Estimate a measure on the number of detected defects/KLOC, due to level of quality of requirements specifications and functional descriptions.
Testers' Experience Defect Detection	The testers' experience. Estimate a measure on the number of detected defects/KLOC, due to level of testers' experience.
Test Method Defect Detection	The number of detected defects/KLOC. Estimate a measure of the number of detected defects/KLOC per tester, due to the test method used.

Table A2. Important in-parameters for module B in the extended and adapted model.

Input parameters module B	Measure
Programmers' System Knowledge Defect Rework	Consider the characteristics in Table below "Programmer participation in reviews", "Number of years with total system", "Number of years with sub-system", and "Used system in laboratory environment". Estimate a measure on the number of reworked defects/day, due to level of system knowledge.
System Complexity Defect Rework	Consider the characteristics in Table below "Common components", "Sub-system's control", and "Other sub-systems' control". Estimate a measure on the number of reworked defects/day, due to level of system complexity.
Quality of Requirements Specifications and Functional Descriptions Defect Rework	Consider the characteristics in Table below "Documentation status", "Review of documents", and "Faults in documents". Estimate a measure on the number of reworked defects/day, due to level of quality of requirements specifications and functional descriptions.
Programmers' Experience Defect Rework	The programmers' experience as a programmer and with the language used. Estimate a measure on the number of reworked defects/day, due to level of programmers' experience.
Code Structure Defect Rework	The degree to which the code is well-structured and well-documented. Estimate a measure on the number of reworked defects/day, due to level of code structure.
Programmer Resource for Rework	The number of programmers for the sub-system development. Since the rework is performed in a later phase it is not a conflict with the programmer resource for coding in the model. The measure should reflect the average number of programmers used for rework.
Rework Method Rework Rate	The number of reworked defects/day. Estimate a measure of the number of reworked defects/day per programmer, due to the rework method used.
Defect Injection Rates During Code Rework	Estimate the % of defects that lead to new defects.

Table A3. Characteristics for a number of in-parameters in the extended and adapted model.

Characteristics	Measure
Programmer participation in reviews	Important documents for code development is reviewed.
Number of years with total system	Number of years of work with total system, in order to know the purpose, structure etc. of the system.
Number of years with sub-system	Number of years of work with sub-system, in order to know the purpose, structure etc. of the sub-system.
Used system in laboratory environment	The programmer or tester has used the system in laboratory environment.
Common components	The use of common components affects the complexity.
Sub-system's control	The sub-system's control and effect on other sub-systems.
Other sub-systems' control	The degree to which the sub-system is controlled and affected by other sub-systems.
Documentation status	The degree to which important documents (requirements specifications and functional descriptions) are complete, i.e. if important parts are missing.
Review of documents	The amount of review and the appropriateness of reviewers.
Faults in documents	The degree of faults found in important documents (requirements specifications and functional descriptions) after release.
Tester participation in reviews	Important documents for testing is reviewed.

PAPER IV

A Spiral Process Model for Case Studies on Software Quality Monitoring – Method and Metrics

IV

Carina Andersson and Per Runeson

To appear in Software Process: Improvement and Practice, 2006.

Abstract

This paper presents a spiral process model for an iterative case study on quality monitoring conducted in an industrial environment. In a highly iterative project, everything seems to happen at the same time; analysis, design and testing. We propose a spiral process model for case studies, and present a study conducted according to the proposed process. In the study, metrics collected from three software development projects are analyzed, to investigate which characteristics are stable across projects and feature groups of the product. The contribution of the paper is multifold, detailing the case study methodology used with its subgoals and procedures. Furthermore, the paper presents the metrics collected and the results as such from the case study, which give insights into a complex development environment and the trends of the retrieved data. The analyzed data serve as feedback to the project staff to facilitate identification of software process improvement. The data have also been used for defect prediction.

1. Introduction

A clear understanding of the software development process and its activities is necessary to successfully produce quality products, and to manage continuous software process improvement. To develop large software products poses challenges, one of which is a large amount of change requests. A common solution is to move to a more and more iterative development process, where the changes in requirements, design and implementation can be responded to. The iterative development environment, where the job is partitioned into smaller parts that accommodate a repetitive delivery of increments, requires an extensive amount of planning and understanding of the planned and unplanned activities. Metrics and data collection have a potential for improving the understanding and control of this development process [21], while the research methodology of a case study could help us to provide this understanding.

However, studying this iterative environment calls for a corresponding case study methodology. Hence, we have defined a spiral process model for case studies in highly iterative development environments. In this paper we present the process model and a case study following its structure. The study is conducted in an organization, developing software in a product line fashion, with an incremental software development process. The organization develops products according to a process with an extensive number of small iterations. The case study originated from an inquiry from quality management in the organization. The aim was to derive software process improvement proposals, specifically related to the activities conducted in the test process, and to improve the planning and prediction capabilities. Hence, the purpose of this specific study was to assess process and product metrics, and investigate whether we could find any patterns across projects, which first and foremost could be useful to provide further understanding of the test process, but also for prediction purposes. Based on this understanding, software process improvement initiatives for the test process could be identified by the project staff. The case study methodology is well suited in an environment where we have little or no control over the variables, but on the other hand will give insights into a specific organization's development process [17].

Other case studies describing measurements and data collection, and specifically focusing on metrics from the test process, have been published; e.g. [5][8][13][18][24]. Often, these case studies focus on

modelling the test process, and prediction of the number of faults or other quantitative analyses. Stikkel has recently presented a model for the test process for estimation of the test effort [24]. This paper provides several views of the data collected from three projects. They are used for validating the model. Levendel presents data from a large, incrementally developed software system, used to model defect detection [18]. Kanoun et al. propose a method to control the test activities based on descriptive analyses and reliability models, which are applied to failure data collected from a switching system [13][14].

In this paper, we focus on the case study as such, describing the chosen methodology and the interaction between the researchers and the studied organization and its staff. To validate the methodology, we present a case study where the test process in a highly iterative development environment has been investigated. Hence, the contribution of the paper is multifold, separated into two parts, methodology and case study application.

In the methodology part, we present the spiral process for iterative case studies. The process aims at guiding a case study with the purpose of basic understanding and more advance identification of process improvement areas. It is further focused on establishing interfaces between the researchers and the project staff at relevant points in the case study process.

The case study, which is conducted according to this process, characterizes the test process of the development organization, using measurements which primarily come from the organization's failure report database. The case study part includes experiences from presentation of the collected and analyzed data, and the interpretation and feedback from the project staff. One specific means of capturing the test process characteristics is the development and evaluation of a defect prediction model.

The paper is organized as follows: In Section 2, the case study methodology with its cycles and steps is presented. In Section 3, the study setting, i.e. a characterization of the investigated organization, is presented, along with the organization's development process and the examined projects. In Section 4, the case study data and its data filtering procedures are described. Section 5 presents the data analysis and discusses the results obtained from the analyses conducted in the different case study cycles. Finally, conclusions from the study are presented in Section 6.

2. Case study methodology

According to Yin, a "case study is an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between the phenomenon and context are not clearly evident" [25]. From the definition, it is clear that the research strategy in a case study must be flexible [22] and of an iterative character to enable adjustments to the studied phenomenon. An overall research direction is set out, rather than specific research questions. The scope of the study is adjusted continuously, as the boundaries between the observed phenomenon and the context are gradually better understood. Further, as the phenomenon is studied in its real-life context, the observational methods should be as non-intrusive as possible.

2.1 The spiral case study process

A highly iterative software development organization is a very complex, contemporary phenomenon. There are no clear boundaries between technical and organizational issues. However, current case study methodologies do not sufficiently define how to conduct such a study [25]. Thus, a new case study process was needed, to be able to address the issues adherent to the situation. The case study process has to have an iterative character, to stepwise adjust the goal and scope to the iterative findings. In addition, the case study process has to manage the interface between researchers and project staff, and explicitly show who have responsibility for the specific parts. Hence, we developed a spiral process model for iterative case studies, which also takes into account different actor's roles in the process. The process is outlined in Figure 1, where the responsibility is shown as white for the researchers, light grey for joint responsibility between researchers and company, and darker grey for the company.

The intention of the proposed case study process is to start with a broad scope, shown as an *exploratory cycle* in Figure 1, and thereafter delimit the scope and adjust and reformulate new goals as more information is available. After exploratory case study cycles, a more *confirmatory* approach is chosen. In this type of cycle, the trends observed in the exploratory part of the case study process could be formulated as hypotheses and tested. After that, the process continues by taking an *explanatory* approach. In the explanatory cycles, explanations of the

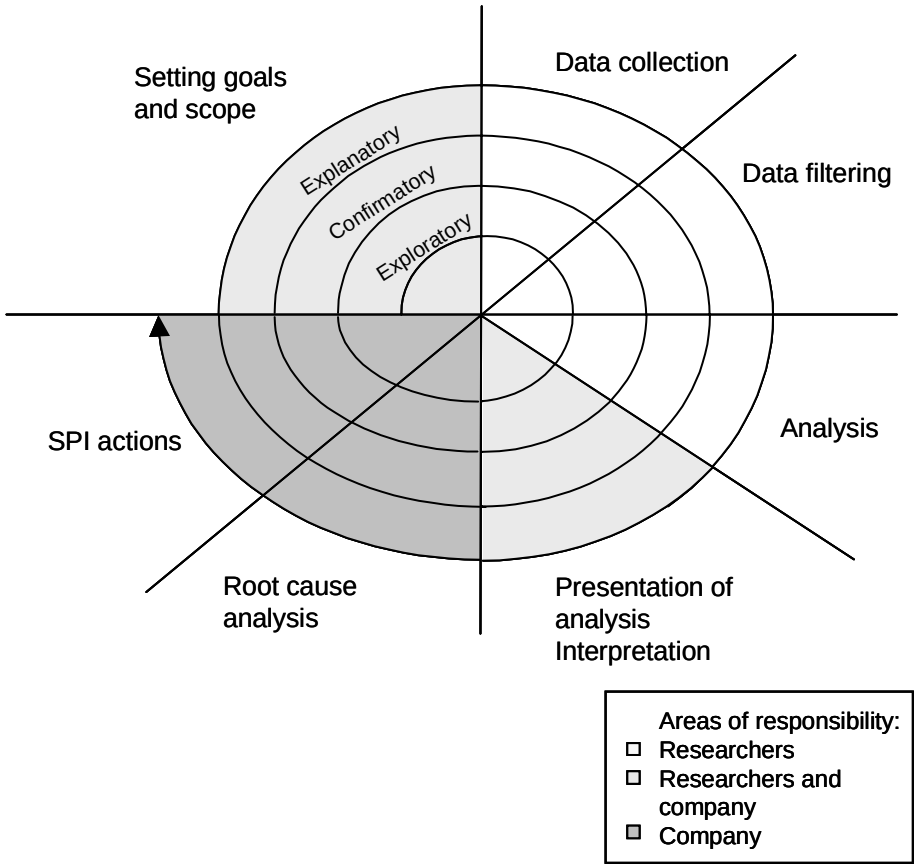

Figure 1. *The spiral case study process.*

behavior observed and hypothesis tested in previous cycles are sought. One type of cycle, whether it is exploratory, confirmatory, or explanatory, could result in several cycles, before moving on to the next type. However, in Figure 1 only one cycle of each type is shown to increase the readability.

Each cycle of the case study process consists of several steps. The first step involves setting up the goals and scope of that specific cycle. This step is performed in close cooperation with the industry participants, to ensure the relevance of the defined goal.

The second step of the case study process involves collecting data, to gain knowledge about the studied object defined in the goal from the first case study step. The data could be of both of quantitative and qualitative type. The real-life context in our study is a very competitive market-driven development organization, hence the case study observations are

not permitted to interfere with the project activities more than absolutely necessary. The roles involved in the study are presented in Figure 2. The researchers interfaced primarily with the project quality assurance person (QA), who in turn was the interface with the project staff. The main reason for this division of roles was that the project should be disturbed as little as possible. The QA person had the responsibility for quality management issues, while the project staff had their focus on product deliveries. Archival data could be accessed directly by the researchers.

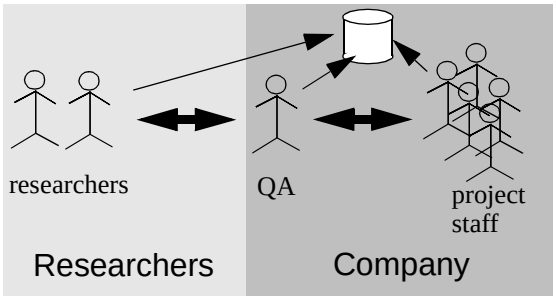

Figure 2. *Roles in the case study.*

In the third step in the case study the researchers, to improve the quality of the data, filter the collected data. Confounding factors are removed from the data.

In the fourth step, the researchers analyzed the data. The observations were presented as descriptive analyses in the earlier cycles, and statistical analyses and hypotheses test were conducted in the confirmatory cycles.

The purpose of the fifth step for the researchers is to present the findings to the company, either represented by the QA or directly to the project staff. The analyses are fed back to the project during these meetings, and the researchers participate in the discussions of the results as it evolves and interpret the data in terms of project events and improvement needs. The presentation step is followed by steps in the case study process that have to be conducted by project staff. Root cause analyses of the presented observations have to be accomplished by persons with in-house knowledge. The same goes for the last step of the case study process. The findings of the root cause analyses create the enhanced understanding that is necessary for suggestions of software process improvements. The initiatives for software process improvements have to originate from within the organization, and also be managed from there.

2.2 Case study process application

When applying the proposed case study process to this specific study, it resulted in a case study with seven main cycles, reported in Table 1, and referred to in the analysis in Section 5. However, we only report on the steps accomplished by the researchers, and when possible briefly mention the actions taken by the company, since the last two steps in the case study process are managed from within the company. This case study process was followed during a period of 12 months, with cycles of 1-2 months.

Initially, when launching the case study, the scope was very broad. The goal was to develop a software process simulation model [16]. The purpose of the simulation model was to analyze the test process, similar to the study by Berling et al. [4]. Unfortunately, the goal to build a model based on mostly quantitative information was not achieved. The scope

Table 1. Characteristics of case study cycles.

#	Goals and scope	Data collection and filtering	Analysis and presentation	Interpretation and improvement
1	Simulation model	Process models Time reports Failure reports	Build simulation model	Too complex approach
2	Exploratory	Failure reports project 1, 2	Distribution of detection activities over time	Response on specific events in each project
3	Exploratory	Failure reports per feature group	Distribution of detection activities per feature group	Motivation for distribution
4	Confirmatory	Failure reports project 3	Same as in cycle 2 and 3	Sufficient fit for practical use
5	Explanatory	Qualitative data on feature groups Subset of failure reports	Characteristics of feature groups	Root cause analysis on causes and suggestions for each group
6	Explanatory (Prediction)	All failure reports	Prediction of defect content with simple model	Use of prediction model to improve planning
7	Explanatory (Prediction)	All failure reports Time data	Software reliability growth models	Use of prediction model

turned out to be too large, since the available information at that time was not adequate for building the model on the required level of abstraction, and the goal has to shift towards a more general analysis of the process, based on what could be collected from the company's failure reports database and other archival sources.

Hence, a second cycle of the case study was launched, with an exploratory goal. Quantitative information, specifically in the shape of failure reports, was collected from two projects to gain knowledge about the studied object. The researchers were given access to the company's intranet, and could thus search the failure reports database directly. The obtained defect distributions were analyzed over time and the behavior was analyzed with respect to specific events and test activities. The data filtering step involved e.g. investigation of by which test activity the failures where observed, which could be cross-checked by mapping the reporting engineer to the test activity he or she was scheduled for.

In the third cycle, the same failure reports as in the second cycle were further analyzed. A new attribute, the feature in which the defects was located, was used as basis for the analysis. The approach was still exploratory, and the analyses were descriptive in the form of bar charts. The findings were fed back to the organization during the presentation step of the case study process and discussed and expanded with qualitative information on the obtained defect distributions.

The fourth case cycle used a confirmatory approach. Quantitative information from a third project was collected. The defect distributions obtained in cycles 2 and 3 were compared to the failure reports from the third project, to investigate whether the observed trends from the previous cycles were found here as well. The analysis and interpretation were mainly based on statistical analyses and hypotheses tests.

The fifth case study cycle took an explanatory approach. This cycle can in reality be separated into several sub-cycles. Qualitative information was collected to explain the behavior of, and to characterize the feature groups. The qualitative data were complementary to the data obtained from the failure reports database and used for explanatory purposes, e.g. to interpret data fluctuation properly and to clarify differences between the projects.

The two last case study cycles were also explanatory, with the purpose of finding prediction models that could explain and predict future behavior. To start with, a simple prediction model of the number of failures to expect in a project was defined, based on the defect

distributions observed in previous cycles. Afterwards, software reliability growth models were applied to the defect distributions, and these models' ability to give appropriate predictions were evaluated and compared to the more simple approach from case study cycle 6.

For the company, the purpose of the last two steps in the case study process is to do root cause analysis of the observations and identify potential software process improvements. As researchers we can only take an observational part [3][22] in the discussions during these case study step. An example of actions taken by the company in one of these cycles was the analysis of a subset of failure reports, using root cause analysis, to identify causes and thereby improvement possibilities. The results from these steps under the responsibility of the company are used for setting the goal and scope of the following cycle.

2.3 Validity

In a case study the threats to validity can be discussed in terms of *validity* and *generalizability* [22]. With a research strategy where the focus is on a case, a sufficient degree of generality can be obtained by the insights given by the data. Then these insights can be projected to other contexts. However, beyond this specific setting we do not see any transferability of the specific defect distributions. The intention is instead to show how data from a complex iterative development environment could be utilized and presented, and for this purpose, the spiral case study process involves feedback from the company participants in each cycle. Further, the gradual movement from an initial exploratory focus, via confirmatory to explanatory focus, increases the validity of the findings.

Concerning validity of the study, Robson summarizes three kinds of threats: *reactivity, researcher bias* and *respondent bias* [22]. Reactivity refers to whether the object under investigation behaves differently due to the presence of the researcher. Researcher and respondent biases refer to whether researcher or respondent bring their backgrounds and personal viewpoints into the study, and thereby color the results. Six strategies to deal with the threats to validity, all applicable in the proposed case study process, are presented. For researcher bias threats, *triangulation, peer debriefing, member checking, negative case analysis* and *audit trail* are actions taken to reduce threats. *Prolonged involvement* may reduce the reactivity and respondent bias threats, but increases researcher bias threats. Actions taken in this study to reduce threats are presented below.

Triangulation is used in terms of data triangulation and observer triangulation. Data triangulation was conducted primarily by data collection from the failure reports database, but also through discussions with representatives from the organization. Observer triangulation was achieved by two researchers and one project QA person working together with different roles during the study, thus enabling peer debriefing and negative case analysis.

The data have been fed back in various formats to the representatives from the organization at several occasions, as member checking, during the presentation steps of the case study process.

Audit trail is used by keeping a record of the activities while carrying out the study, ranging from collection of raw data, details of the data filtering, to data analysis and interpretation. The data filtering is considered to be one of the most important steps in the data validation stage, starting from cleaning the raw data to eliminating confounding factors, to assessing the quality of the attributes of every data point.

The length of the case study, and the characteristics of the case study process with continuous feedback and discussions with company representatives have reduced reactivity and bias from the project staff through prolonged involvement. The threat of greater researcher bias with prolonged involvement is reduced by the steps in the case study process where the company representatives have the responsibility of doing root cause analysis and software process improvement actions, without interference from the researchers.

In summary, relevant actions are taken to improve the validity of the study, some of which are inherent in the spiral case study process. The conditions specific to this particular case influence the outcome. However, given the extensiveness of the data collection we believe that there are facts of interest to a wider community.

3. Study setting

The case study presented in this paper was conducted in an organization developing consumer products with an extensive list of features, as requested by the market. The products are developed using a product-line fashion [6]. They are implemented on a hardware and software platform, delivered by a subcontractor. Internally within the organization, a software platform is shared among different products, thus following a

product line concept. Software modules may appear in different products, either as general components or as a specifically tailored version of the component for the specific product.

3.1 The development process

The development process used in the participating organization is an incremental process, with component releases in small iterations. The typical project duration is in the range of a number of calendar months. The product to be developed in a project is decomposed into parts, each part consisting of a subset of the product's features. A subgroup of project members, called a *feature group*, is responsible for each part of the system, i.e. each feature group delivers a specific subset of features of the software requested. Each feature group consists of a number of developers, dependent on the size of the assigned part, and each feature group also has its own function testers, typically in the proportion of one tester for two developers. Thus, function testers have defined roles, different from the developers. The developers conduct unit testing themselves. After that, the function testers take over the testing role. Each feature group bases its design and implementation on the main software product. After function testing, the feature group delivers the increments of its assigned part to the main product, see [3]. The figure illustrates how the main product grows incrementally as each feature group is delivering new code.

The system test organization is separate from the development and function testing feature groups. The system test organization runs its test

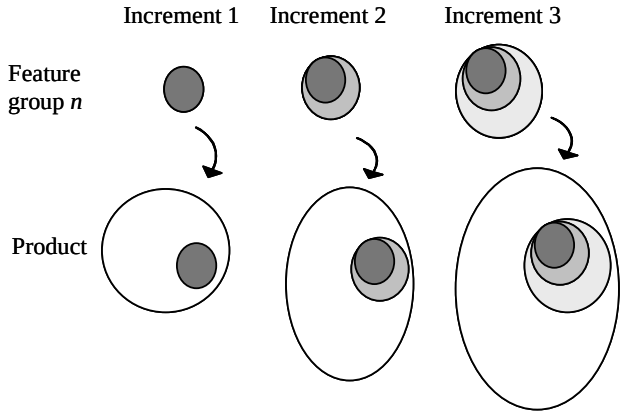

Figure 3. *The incremental development of a product, illustrating the functional growth of one of the feature groups.*

suites on the latest available version of the product. The testing is performed both in the development environment and in the real operational environment.

In addition to function test and system test, customer acceptance test is conducted in the development process. Dedicated staff conducts this activity separately.

The project management model used in the company is of stage-gate character [7]. This model has distinct phases of pre-study, feasibility study, execution and completion phases. After each phase, there is a gate, at which the project is reviewed with respect to project progress and market opportunities. Further funding of the project is decided upon based on the review. Milestones are set to match the gates, but there are also milestones for intermediate checkpoints.

Using a stage-gate model does not imply using a waterfall approach to the technical development process. On the contrary, parallel and iterative development processes can be used together [15]. In this company, milestones are defined in terms of the status of the product, rather than in terms of development phases. Milestone *Alpha* relates to a product with all new features added. Milestone *Ship* relates to a product that is ready to be released to the customer for formal acceptance test. An illustration of the workflows in the development process is given in Figure 4. Distinct phases of the technical development process are difficult to define in this particular company, since the activities all run in parallel and usually overlap. Nevertheless, Figure 4 gives an overview of some of the

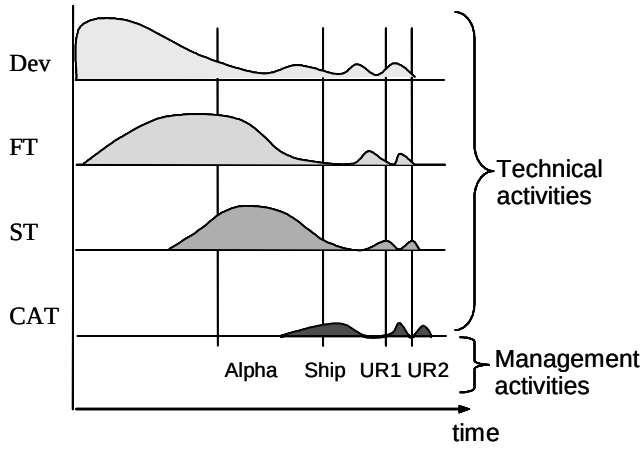

Figure 4. *Workflows related to process milestones. Dev: development, FT: function test, ST: system test, CAT: customer acceptance test.*

milestones used in the projects. The dates for the milestones used in a project are decided on at project start by project management. Most often these are not negotiable. As can be seen in Figure 4, before the milestone *Alpha,* development and function test is the primary occupation, thereafter system test runs in parallel with the feature groups' bug fixing and continued function testing. Also, several regression test periods are included in the process. These are conducted as "campaigns", comprising both function and system test. A second milestone of major interest in this study is the *Ship date*, when the software is released to the main customers in its first version. After the ship date testing still continues for a while, and even further development occasionally occurs. These later corrections and implementations are released in later versions of the software, so-called *update releases* (URs).

A failure reports database is used for tracking of the evolution of the software, i.e. to control the process and quality of the product, e.g. through the analysis of graphs presenting reported failures per week and analyzed failure reports per week. Detected failures should always be reported to the database. However, in unit test, which are conducted by the developers themselves, reporting only happens occasionally, since these defects often are corrected immediately. Failures detected by others than the developers are all reported to the database. The detected failures are reported as *incident reports*, and thereafter analyzed by a change control board (CCB). This analysis results either in a *task report* describing the failure, or a rejection of the failure, i.e. the incident report is not considered to describe a real failure or it is proven to be a duplicate of a previously reported failure. The task report is assigned to a developer having responsibility for the correction of the defect. The failure detection process is illustrated in Figure 5. A task report has a number of characterizing attributes, listed in Table 2. Some of those attributes are used for classification of the defects included in the case study, which is further described in Section 4.2.

To summarize, the development and testing processes are highly iterative and feature driven. The process and its environment are fairly well established in the organization, and have been used and maintained over several projects.

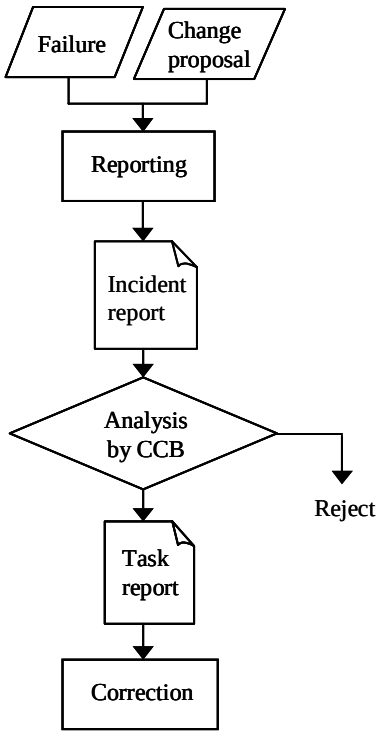

Figure 5. *Failure handling process of the studied company.*

Table 2. Attributes of a task report.

Attribute	Description
Task No.	Identity of the task report
IR No.	Cross-reference to the underlying incident report
Description	Description of the observed failure, given in natural language
Date found	When the failure was observed
Reporter	Identification of the person who observed the failure
Module	Location of the defect
Status	Status of the task report
Severity	Severity of the defect

3.2 The observed projects

The data presented in this case study come from three successive software development projects. For the presentation in this paper the data have been rescaled, to facilitate the comparison between the projects, specifically the total size of the projects, the number of failures and the projects' length, which anyhow on a detailed level is not relevant for the research purpose.

The application domain is the same for the projects, though still several characteristics for the projects differ. Figure 4 gives an overview of the two major milestones (Alpha and ship date) in the technical development process, which are used for all projects. The lead-times, calculated from the start of the projects until ship date according to the figure, are in the scale of 10:13:9, respectively for the three projects. The size of the projects also differ; project 1 is only half of project 3 measured in lines of code, while project 2 lies in between. The complexity of the code is not measured and hence not used in the analysis. The total numbers of failure reports included in the analysis are in the scale of 10:25:15 for the three projects, respectively.

The three projects consist of the same feature groups, with small variations. An overview of the project characteristics is given in Table 3.

Table 3. Project overview.

	Duration (scaled)	**Size (scaled)**	**Number of reports (scaled)**	**Number of feature groups**
Project 1	10	10	10	10
Project 2	13	15	25	12
Project 3	9	20	15	12

4. Case study data

The case study was, as presented in Section 2, launched using the spiral case study process model, starting each cycle with setting goals and continuing with collection and analysis of quantitative information [22]. In this paper we focus on presenting the descriptive analysis of the data

collected from the failure reports database. Company representatives used the result of the analysis to identify software process improvements.

Common assumptions for the projects were stated regarding defect distributions at several different points in time. These points could be milestones or other project events, such as regression test periods. The exploratory analysis of the data in the first cycles of the case study process was for the later stages of the case study followed by a more confirmatory approach to establish whether our assumptions of similar defect distributions between projects could be confirmed or not. Finally, the explanatory findings were used for prediction purposes.

4.1 Goal and scope

The overall goal of the case study was to understand the current test process with the intention of improving it. Improvement goals include reduced cost, increased quality, and improved planning and predictability.

The scope of the case study was framed iteratively during the cycles, but it was limited to the test process, including the test activities function test (FT), system test (ST), and customer acceptance test (CAT). Development activities such as design, implementation and unit test are excluded, since failures detected in these activities are not reported on a regular basis. However, additional activities produce failure reports, comprising failures reported later on in the project, e.g. by project management and others using the software in pilot use. These are included and marked miscellaneous (Misc.).

The scope of interest is limited in time to include failures reported from project start until the date set for shipping, *ship date*, see Figure 4. Due to the product line development used in the company, one component could be used in several products, and new functionality could be added after this date, and then delivered in the update releases. However, the frequency for new or changed functionality after the ship date varies from one project to another and from one feature group to another, thus creating large differences between the projects. By limiting the scope in time to the ship date, we achieve a more predictable situation across projects.

4.2 Data collection and filtering

In the stages of data collection in the case study process, data were extracted mainly from the failure reports database. The first data filtering involved the query used for data extraction, which specified that only task reports were included, i.e. the incident reports had gone through a first analysis and were considered to be real failures. Task reports regarding the external platform are not included. Change proposals are filed as change requests and not as failures, hence, these were excluded from the analysis. Duplicate failures are excluded, since these failures are not recorded in the failure reports database as new task reports when they are rediscovered. If the cause of a detected failure was corrected by modifying *n* modules, it is counted as *n* distinct defects. Even if the specific correction in a module concerns several lines of code, this counts as one defect.

The next step of the data filtering included classifying the task reports, from here denoted as defects, by using some of the attributes in Table 2: *date found, reporter, module,* and *status*.

Date found specifies when the defect was detected, i.e. the date when the failure was exposed by testing and consequently the incident report was filed. The precision used was per day. *Reporter* identifies who detected the defect. All reporters were classified according to which activity they belong to (function test, system test, customer acceptance test or miscellaneous) and for the function test activity, they were also classified according to which specific feature group they belong. For project 3, the reporters themselves clarified in the incidents reports, whether they detected the failure in the role of function tester, system tester, customer acceptance tester or miscellaneous. Their own classification was compared with the classification conducted by the researchers, to ensure that the two classifications agreed. *Module* identifies the software module in which the defect was located. The classification of defects per feature was based on the modules, since a certain set of modules is part of the functionality under responsibility of a specific feature group. *Status* identifies which status the defect had at the extraction date. Thereby task reports rejected, cancelled or not repeatable were excluded from the analysis.

The severity of the defects is not included as an attribute for classifying the defects in the analysis, i.e. all reported failures are treated equally, regardless of criticality. This approach is chosen since discussions indicated that all defects were important for planning of the proper effort to correct them. Thus the analysis of each defect is performed regardless

of its severity. In later stages of the analysis a new perspective of the data in the form of defect severity could be of interest, although this is not yet implemented.

5. Data analysis and results

The data were presented to the representatives of the company as part of the feedback inherent in each cycle of the case study process. Its purpose was to motivate the company representatives to continue the analysis of the data within the organization, in terms of root cause analysis. This would in turn support them in identifying software process improvements possibilities. During these feedback meetings different views of the data were examined to find the most valuable data formats for this organization. The data were presented both in the shape of graphs for the entire projects as well as separately for each feature group. Some of the views are presented below. Cycles 2 and 3 in the case study explored projects 1 and 2 in terms of defect distributions over time for the whole projects and separated for the feature groups (see Table 1). In the fourth cycle of the case study, which was confirmatory, project 3 was examined by the same means. Thus, the presentation below includes all three projects, even though the analysis was conducted during different case study cycles.

The data are primarily given in three dimensions, based on *time* (date found for the defect), and *test activity* (based on which activity the reporter belonged to), both dimensions are derived from case study cycles 2 and 4, while *feature group* (which functionality was affected by the defect), is derived from case study cycles 3 and 4. The milestones, and other scheduled events which were common for the projects, were used as reference points of time when exploring the data for similarities.

5.1 Defect detection over time

5.1.1 Entire project

The first exploratory case study cycle (cycle 2 in Table 1) resulted in a presentation format that was used for describing the current state of the projects, see Figure 6, in terms of *Number of defects detected over time*. To

give an overview of the distributions of total defect detection, we show the number of defects detected per month over time. The distributions reflect defects reported from project start to project end, reported by function test, system test, customer acceptance test and miscellaneous. The three projects were compared at specific dates. The ship dates are indicated in the figure, but other milestones following the technical development process that are comparable in the projects were also used.

As can be seen from Figure 6, the shape of the time distributions differ between the projects. The appearance of the defect distributions illustrated in Figure 6, and more specifically distributions for each test activity over time, were discussed and explained by the project staff during the feedback meetings. The response from the project participants was information about specific events that explain why a certain maximum is reached at a certain point, when the test activities are more intensive. An example of an event is when function testers run a regression test period. An unexpected minimum could be due to Christmas, when most employees take some days off. Hence, limited general conclusions could be drawn from this view of the data.

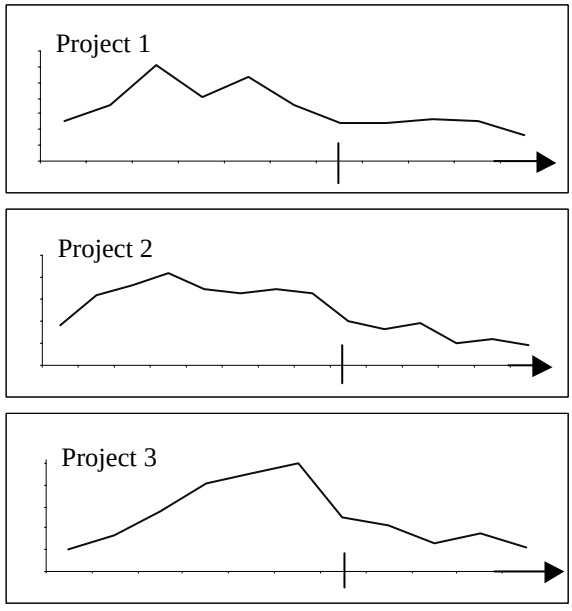

Figure 6. *Defect distributions, showing number of defects detected over time, for the three studied projects. Ship dates are indicated with a vertical bar.*

5.1.2 Test activity

The first exploratory case study cycle (cycle 2 in Table 1) also included an analysis of the distributions of defects found over time when split into the test activities: function test, system test, customer acceptance test and miscellaneous. The distributions of defect detection per month in the different test activities were examined, up to the ship date. The percentage of defects detected in each test activity compared to the total number of defects detected at ship date, in each of the three projects, are given in Table 4. By truncating the data sets by the ship date, dissimilarities between the three projects were decreased, and the three projects could be compared without regard to the project decisions on update releases made after ship date. The project decisions originate from the continued development and testing which is performed due to the product line development. These decisions were very project specific, sometimes resulting in new functionality development, at other times not, making it difficult to draw any conclusions between the projects after this date. This truncation also eliminated any possible confusion as to which project a defect belonged. For some feature groups the development continued in a subsequent project, using the same modules, i.e. certain components were part of more than one product in the product line.

The distribution of defects found by the test activities appeared to be quite stable across the three projects, as can be seen in Table 4. However, a χ^2-test [23], with the defect distributions from three projects as samples does not show on any statistically significant similarities at a significance level of $\alpha = 0.05$.

In addition, other milestones in the projects were used as basis for comparisons. Specifically the dates for finished regressions test periods

Table 4. Percentage of defects detected by the different test activities before ship date.

Test activity	Project 1	Project 2	Project 3	Average
FT	67%	69%	62%	66%
ST	19%	25%	30%	25%
CAT	5%	2%	3%	3%
Misc	9%	4%	5%	6%

Table 5. Percentage of defects detected in *function test* at Alpha, compared to ship date.

Project 1	Project 2	Project 3	Average
80%	81%	80%	80%

reflected commonalities between the projects. Until the first regression period, mainly function testers have reported defects. System test starts before Alpha, although the main part of the resources is spent after this point in time. What happened until this point in time, equivalent to milestone Alpha, was very similar between the projects. In Table 5 the number of defects detected by function test until the milestone Alpha, divided by the total number of defects detected by function test until the ship-date for the projects, is shown. This ratio is quite stable over the three projects, and in the χ^2-test the hypothesis of no difference between the samples could not be rejected.

The response from the project staff to this data in terms of process improvement was typically discussions on what should be considered an optimal distribution of defects detected over phases. There were strong arguments for moving as much as possible upfront to function test. However, as function test and system test are conducted in parallel in some stages of the project, the picture is not clear. However, reducing customer acceptance test problems to a minimum, was an agreed upon desire. The distribution across the activities was further analyzed for each feature group in the next case study cycle.

5.2 Defect detection per feature group

In the second exploratory case study cycle (cycle 3 in Table 1), the defect data were classified and analyzed by individual feature group. The analysis showed that at this level of detail, the variations between the projects were too extensive to draw any conclusions for each individual feature group. This was confirmed by statistical tests. For some groups, trends could be observed and the pattern was similar for the three projects, while other groups showed no such behavior. The defect distribution for a subset of the feature groups in project 2 is presented in Table 6, where the last row shows the average for the whole project, i.e. including all feature groups in the project. The data for each feature group reflects the differences between the groups, while on a higher level of abstraction the average for

Table 6. Defect distribution for a subset of the feature groups in project 2.

Feature group	FT-own	FT-others	FT-all (own + others)	ST	CAT	Misc.
1	74%	14%	88%	6%	0%	6%
2	69%	6%	75%	20%	3%	2%
3	35%	29%	64%	26%	1%	9%
4	23%	41%	64%	26%	2%	8%
5	64%	11%	75%	20%	0%	5%
6	13%	31%	44%	40%	8%	8%
7	69%	19%	88%	9%	1%	2%
8	42%	6%	48%	52%	0%	0%
Entire project			69%	25%	2%	4%

the three projects, previously discussed and shown in Table 4, are more similar. The figures for the *entire project* are calculated based on the absolute number of defects for each feature group in the project, i.e. including the groups not presented in Table 6.

In Figure 7 defect distributions for function test and system test for a subset of the existing feature groups is presented. Variations within groups between projects could be seen for several of the feature groups. Some trends could be noticed. Feature group number 4 has a similar defect distribution in function test over all three projects. However, when analyzing the distributions for system test, no trends could be noticed for the same group. Note the order of projects, for the function test chart the order is project 3 in front of project 2 and project 1, while for the system test chart the order is reversed to improve readability.

The analysis also examined whether function testers who reported defects in a specific module, were responsible for testing these modules, i.e. whether they belonged to the tester's own feature group, or whether the defects were located in modules, which were part of other feature groups' functionality. The column *FT-own* in Table 6 shows the percentage of defects detected by function testers in the own feature group compared to the total number of defects detected for this feature group. Column *FT-others* shows the percentage detected by other

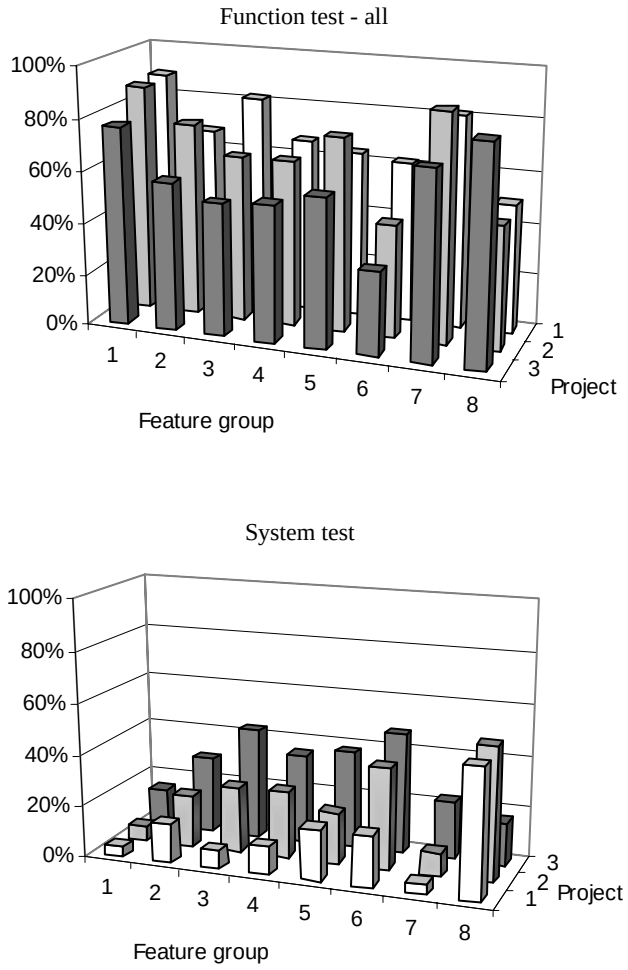

Figure 7. *Defect distribution displayed for function test and system test for a subset of feature groups. Note that the order between projects is switched between function test and system test.*

function testers, i.e. these testers belonged to other feature groups. Column *FT-all* is the total of FT-own and FT-others added together.

The feedback to this data included two interesting aspects. First, a tendency of competition between feature groups arose in the discussions, awarding those with high FT percentages. However, later, extreme values in Table 6 could be explained by specific behavior in the feature groups. As an example, feature group number 1 has responsibility for a coherent code base, which is not used much by other groups, and also tested in a

rather extensively automated test environment. On the other hand, feature group number 6 develops functionality with a high degree of interaction with other applications. Second, the distribution between FT-own and FT-others was a surprise to the researchers, while the company did not see any problem in which of the test teams find the defects belonging to a specific feature group.

In the following case study cycle further analysis of the feature groups was performed, to explain the observations.

5.3 Feature groups' characteristics

In case study cycle 5, the analysis of each feature groups' characteristics gave the feature group leaders an insight into his or her own group's performance, compared to the others. This analysis involved searching for a best testing practice among the feature groups. However, it is important to notice that the number of defects detected by a group depends on several factors that are external to the group. Some groups, which are more dependent on others, may not expect as high percentages as more independent groups, since the independent groups may not have functionality closely related to others or they do not have to wait for functionality to be implemented by other groups. No conclusions on this feature group level of detail could be used for prediction of future projects, since the performance of the feature groups differed from one project to another.

The analysis of defects detected by the feature groups' own function testers compared to those detected by other function testers in Table 6 was of interest to us as researchers, since no rule of thumb have been found regarding what could be expected. However, from the organization's perspective this was of less interest. Which function tester detected the defect was of less concern. The ability to detect the defects in one of the other feature groups' test cases was no surprise, since specifically at the regression test periods, everybody starts running their test cases at the same time, and some functions are closely related to each other through interfaces. In this situation a defect may be detected by another group before the own group has reached a test case revealing it. A high defect detection rate for the own function testers was the result of a more extensive test plan or more automated tests or less interaction with other features. Meanwhile, a low defect detection rate for a feature group's function testers could originate in a code base more used and adapted by

other groups, but also in a smaller amount of resources and decisions giving priority to other activities than function test.

In the case study process step under the responsibility of the company, they conducted e.g. root cause analysis of a subset of defects. Thus, the causes and stages of defect injection were surveyed and clarified, to facilitate identification of improvement areas.

5.4 Defect prediction with simple model

In case study cycle 6 the goal was to improve the company's defect prediction approach with a simple model. Fenton and Neil classified defect prediction models into four categories [10]:

- prediction using size and complexity,
- prediction using testing metrics (defect rates),
- prediction using process quality data (e.g. CMM), and
- multivariate approaches.

Fenton and Neil conclude that by applying one model or another, the models often produce different results, even for the same data set. A lot of effort is required to find the model that gives the most reasonable prediction with high accuracy. A second aspect of the suggested models is that most published studies present how the application of the models has been conducted after the projects are finished. Thereby, the results of the applied models could be validated, but on the other hand it makes it difficult to evaluate the models' applicability early in the process, which most often is needed in practice.

A further problem of the existing models is that these often assume a well controlled environment with few increments. The models are based on "early" and "late" phases. In highly incremental processes, like the Rational Unified Process (RUP) [12] or even more in the eXtreme Programming (XP) context [2], the technical development activities are not performed in sequential phases of, for example, requirements engineering, design and testing, but rather totally in parallel. This extreme parallelism changes the notion of the traditional phases, like design, development and test, and hence the applicability of prediction models, which are based on a phase concept. Note that the project management activities still have some kind of a phased approach even when the

technical development process is highly iterative, see Karlström and Runeson [15].

Since defects were classified according to the role of the defect reporters instead of according to the (non-existing) development and test phases, we developed a tailored prediction model, based on the defect distributions presented in Section 5.1. The purpose of the model is to be applied in future projects at milestone Alpha to predict the number of defects to expect to be found up till ship date. Quantitative information from the analyzed projects shows that the number of defects reported by function test, when reaching the milestone Alpha, are 80% of the total number of defect reports reported by function test until the ship date, regardless of the size and lead-time of the projects, see Table 5. The distribution of defects found in different test activities (function test, system test, customer acceptance test or miscellaneous) is also very similar for the three analyzed projects, although not statistically significant, see Table 4, and thereby the total number of expected defect reports at the ship date ($X_{Total,Ship}$) could be calculated, as follows.

$$X_{Total,Ship} = X_{FT,\alpha} \times \frac{1}{Share_{FT,\alpha}} \times \frac{1}{Distr_{FT,Ship}} \quad (1)$$

where

$X_{FT,\alpha}$ = Total number of defects detected by function test at Alpha,

$Share_{FT,\alpha}$ = Percentage of defects detected by function test at Alpha compared to ship date,

$Distr_{FT,Ship}$ = Percentage of defects detected by function test compared to system test, customer acceptance test and miscellaneous at ship date

Using the data from this study would result in the following total number of defects to expect at ship date assuming 100 defects have been reported by function test at Alpha:

$$100 \times \frac{1}{0{,}80} \times \frac{1}{0{,}66} \cong 189 \qquad (2)$$

The total number of defects could be separated into the expected number of defects detected by each individual test activity (from Table 4).

The purpose of the proposed defect prediction approach is not to formulate a new mathematical defect prediction model, but to show that

in this case, some common sense and simple rules of thumb based on the observed defect distributions are adequate. For the investigated company this approach was shown to serve a useful purpose. The defect prediction is based on data obtained from three projects where the defect distributions over testing activities have been fairly stable. This is a case study of one particular organization and we claim no applicability of the exact figures to other organizations' test processes. However, related research by Fenton and Ohlsson [11] supports the approach by showing that software products developed in similar environments have similar defect densities in similar testing activities. This makes it possible to replicate the approach in other organizations, provided that the process followed is relatively stable. Differences between projects should not be too substantial, to be able to receive estimates with some accuracy.

5.5 Additional case study results

As described in Section 2.2, the first cycle in the case study concerned a software process simulation model. The goal was to develop a simulation model, by adapting a previously developed template model [4], and qualitatively evaluating its impact when introduced in the organization. As it turned out, not enough information was available for developing an accurate and useful simulation model with sufficient level of detail. For example, test effort was an aspect of interest. However, the metric test effort separated by activity and feature group is not defined and collected in the organization. Hence, we could not develop a more detailed model at a lower project-level. On the other hand, when investigating the projects at a higher level of abstraction as in this study, similarities are observed and conclusions can be drawn, although this was not what we had originally in mind. Nevertheless, the company's representatives were satisfied with the higher abstraction level analysis of the project data and found the results valuable.

During the data filtering in case study cycles 2, 3 and 4, the data sets were truncated at ship dates to obtain similar conditions for all projects. Although not presented in this paper, an analysis of the behavior of the defect distributions after ship date was conducted. One of the projects had an extensive amount of new functionality added after shipping, to be released as update releases. The other projects did not have the same addition of functionality. While the conditions for the projects were not

comparable after the ship date, the analysis still provided useful information for the individual projects and their management.

The goal of case study cycle 7 is to apply software reliability growth models (SRGMs) [19][20] to the defect data. Some basic assumptions of SRGMs are violated, e.g. system test is not executed according to an operational profile, and the time between failures are measured in calendar time, not execution time as often is required. Nevertheless, the application of the SRGMs is work in progress, with the preliminary results that the application of the growth models to the cumulative defect data over time works well. However, the predictions of total number of defects stabilize rather late in the test process [1]. Unfortunately, the predictions seem to stabilize too late to be of much value for quality management. The simple defect prediction approach described in Section 5.4 was more suitable in this context, since the predictions from that approach could be calculated earlier in the process.

6. Conclusions

The case study reported in this paper aimed at investigating a highly iterative development process. The software development is done in a product-line context, in a feature-driven fashion. In order to study the testing in the company, we defined a spiral case study process with iterative cycles and interaction between the company and the researchers. The research study as such started by a quality manager who sought operational support from quality management, for identifying potential software process improvements, while the researchers where interested in understanding the case as such. Both parties agreed that the both goals were feasible, as the understanding of the process is based on analysis of the obtained information, and software process improvement initiatives in turn are identified as the understanding is increased.

We launched a flexible design case study according to the spiral case study process, consisting of several case study cycles, each having its own goal and analysis. In the first cycle, where our ambition was to develop a simulation model, we got our first setback when discovering that the desired data were not available. On the other hand, the practical expectations from the industry participants did not require the model, but they could see value in the descriptive analyses presented to them in the subsequent case study cycles. Hence, our ambition in these cycles was

guided by the aim of simplicity. Occasionally simpler approaches are enough even if the development process is complex with an extensive number of small increments. To be a reasonable choice for industry application, simplicity often is necessary, and the threshold for knowledge transfer has been high. Information to utilize in the developed prediction model must be easily available at the time the prediction is to be made. It is not an option to use a number of defect prediction models to find out which gave the best result, when input metrics are not yet available in the organization.

The case study methodology resulted in a series of cycles, starting with exploratory cycles, to investigate the data. Thereafter followed more explanatory cycles, which described the current situation and gave input to the identification of software process improvement initiatives. Finally, the then known process and situation could be further analyzed in the predicting cycles, where attempts to predict future outcomes of the projects are conducted.

We found the spiral process model very useful in monitoring software quality issues in a highly iterative development process. The process enabled a fruitful interplay between the researchers, collecting, analyzing and presenting data, and the company representatives, interpreting and finding root causes and process improvement proposals to address the problems identified.

7. References

[1] Andersson, C., "A Replicated Empirical Study of a Selection Method for Software Reliability Growth Models", *Technical report*, CODEN:LUTEDX(TETS-7216)/1-28/(2005)&local37, Dept. of Communication Systems, Lund University, Sweden, 2005.

[2] Beck, K., *Extreme Programming Explained - Embrace Change*, Addison-Wesley, 2000.

[3] Bell, J., *Doing Your Research Projects* (3rd ed.), Open U.P., 1999.

[4] Berling, T., Andersson, C., Höst, M. and Nyberg, C., "Adaptation of a Simulation Model Template for Testing to an Industrial Project", *Proceedings of 2003 Software Process Simulation Modeling Workshop (Prosim'03)*, 2003.

[5] Berling, T. and Thelin, T. "An Industrial Case Study of the Verification and Validation Activities", *Proceedings of International Software Metrics Symposium*, pp. 226-238, 2003.

[6] Bosch, J., *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*, ACM Press/Addison-Wesley, 2000.

[7] Cooper, R. G., *Winning at New Products* (3rd ed.), Persues Publishing, 2001.

[8] Daskalantonakis, M. K., "A Practical View of Software Measurement and Implementation Experiences Within Motorola", *IEEE Transactions on Software Engineering*, 18(11):998-1010, 1992.
[9] Fenton, N. E. and Pfleeger, S. L., *Software Metrics: A Rigorous and Practical Approach*, Thomson Computer Press, 1996.
[10] Fenton, N. E. and Neil, M., "A Critique of Software Defect Prediction Models", *IEEE Transactions on Software Engineering*, 25(5):675-689, 1999.
[11] Fenton, N. E. and Ohlsson, N., "Quantitative Analysis of Faults and Failures in a Complex Software System", *IEEE Transactions on Software Engineering*, 26(8):797-814, 2000.
[12] Jacobson, I., Booch, G. and Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.
[13] Kaâniche, M. and Kanoun, K., "Reliability of a Commercial Telecommunication System", *Proceedings of International Symposium on Software Reliability Engineering*, pp. 207-212, 1996.
[14] Kanoun, K., Kaâniche, M. and Laprie, J., "Qualitative and Quantitative Reliability Assessment", *IEEE Software*, 14(2):77-87, 1997.
[15] Karlström, D. and Runeson, P., "Combining Agile Methods with Stage-Gate Project Management", *IEEE Software*, 22(3):43-49, 2005.
[16] Kellner, M. I., Madachy, R. J. and Raffo, D. M., "Software Process Simulation Modeling: Why? What? How?", *Journal of Systems and Software*, 46(2-3):91-105, 1999.
[17] Kitchenham, B., Pickard, L. and Pfleeger, S. L., "Case Studies for Method and Tool Evaluation", *IEEE Software*, 12(4):52-62, 1995.
[18] Levendel, Y., "Reliability Analysis of Large Software Systems: Defect Data Modelling", *IEEE Transactions on Software Engineering*, 16(2):141-152, 1990.
[19] Lyu, M. R. (editor), *Handbook of Software Reliability Engineering*, McGraw-Hill, 1996.
[20] Musa, J. M., Iannino, A. and Okumoto, K., *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, 1987.
[21] Pfleeger, S. L., "Lessons Learned in Building a Corporate Metrics Programme", *IEEE Software*, 10(3):67-74, 1993.
[22] Robson, C., *Real World Research* (2nd ed.), Blackwell Publisher, 2002.
[23] Siegel, S. and Castellan, N. J., *Nonparametric Statistics for the Behavioral Sciences*, McGraw-Hill, 1988.
[24] Stikkel, G., "Dynamic Model for the System Testing Process", to appear in *Information and Software Technology*, 2005.
[25] Yin, R. K., *Case Study Research: Design and Methods* (2nd ed.), Sage Publications, 1994.

PAPER V

A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems

Carina Andersson, Per Runeson

Submitted, 2006.

V

Abstract

In order to add to the empirical knowledge on fault distributions during development of complex software systems, a replication of a study by Fenton and Ohlsson [6] is conducted. The hypotheses from the original study are investigated using data from an environment, which differ with respect to e.g. system size, project duration and programming language. We have investigated four sets of hypotheses on data from three successive projects in the telecommunications domain: 1) the Pareto principle - a small number of modules contain a majority of the faults; 2) fault persistence between test phases; 3) the relation between number of faults and lines of code; and 4) fault density similarities across test phases and projects. In the replication, 1) the Pareto principle is confirmed; 2) a high fault incidence in function test is shown to imply the same in system test, as well as pre-release vs. post-release fault incidence; 3) the size relation from the original study could neither be confirmed nor rejected; and 4) fault densities are confirmed to be rather similar across projects. Through this replication study, we have extended the knowledge on fault distributions, which occasionally seem to be stable across environments, and thus gradually extending the generality of the hypotheses.

1. Introduction

Fault distributions in development of complex software systems are interesting to analyse, both from a practical and a scientific point of view. For the practitioner they provide support in project and quality management. For the researcher, they contribute to building empirical knowledge in software engineering, which is urgently needed to make the field evolve and be more mature. This paper reports on a quantitative analysis of fault distributions in three different development projects. The analysis, based on rather extensive data collection, gives empirical evidence and continues to build empirical knowledge, instead of relying on anecdotal or intuitive arguments.

Relatively few studies have empirically investigated quantitative issues with fault distributions in the development of complex software systems. Possibly studies are being done internally in companies, although not published. However, more likely they are not conducted at all. Several reasons could be imagined, for personnel inside the organization: too expensive since it is time consuming to analyse the data, and for researchers: difficulties to get access to the appropriate data.

A quite elaborate quantitative analysis of fault distributions is published by Fenton and Ohlsson [6]. They present a study examining faults from two consecutive releases of a telecommunication switching system. The study investigates a range of the conventional wisdom about fault distributions in the software engineering domain. Although very carefully conducted and reported, the study is a single case study, with no claims on generality. It has to be replicated in other environments to provide better understanding on the generality of the results.

The need for replication studies is often mentioned in concluding chapters of empirical studies, whereas few replication studies actually are conducted. To gain confidence in the results, conducting or participating in families of research studies, rather than relying on single studies, is advocated [21]. By replicating a study, more generalizable results may be achieved, rather than isolated findings from a single study [12]. In the domain of experimentation, replication studies exist, e.g. in the area of defect detection techniques [25], but replicated case studies are rare [27]. In the few replication studies conducted within software engineering, the issue of similarity between replicated experiments is treated differently. Miller [14] states about reusing experimental material, “although from a simple replication point of view, this seems attractive; from a meta-

analysis point of view this is undesirable, as it creates strong correlations between the two studies". Pickard et al. on the other hand [22] state "The greater the degree of similarity between the studies the more confidence you can have in the results of a meta-analysis."

The goal of our study is to replicate the above mentioned study by Fenton and Ohlsson [6], and analyse their hypotheses on a different type of system in a different development context. Due to the characteristics of the investigated phenomenon, the hypotheses are not formally tested using statistical hypothesis test, but interpreted qualitatively, based on statistical analyses of data. Their hypotheses concerned four areas, each examined in the study presented in this paper. It is important to continue to build a body of knowledge by further analysis of the areas. However, our goal is not to generalize to the whole software engineering community, because the results as such have shown to be dependent on other explanatory factors. In the empirical picture of the software development process, and specifically the fault detection process, for example the type of software system developed should be taken into account, to be able to draw any valid conclusions. Still, we lean more towards the position by Miller on how a replication should be conducted [14] and intend to vary parameters of size, system type and development process in our replication. The measurements are as far as possible kept the same, whereas some are not possibly achievable in the replication context. Furthermore, it is argued that by deliberate variations in the conditions of the research study, an increase in the external validity is achieved, and hence a higher degree of generalization [12]. By running a replication of the study by Fenton and Ohlsson, we extend the scope of the results to a new environment and investigate the range of conditions under which their findings hold.

The rest of the paper is organized as follows: In Section 2, related work on fault distributions is presented. The hypotheses from the original study by Fenton and Ohlsson are listed in Section 3. The observed organization is briefly presented in Section 4. Our results are presented in Section 5, in particular together with discussions of the results from the original study by Fenton and Ohlsson, but also with related studies when applicable. Section 5.8 discussed the results and conclusions from the investigated hypotheses, while Section 6 presents the conclusions that could be drawn from replicating the original study.

2. Background

Fenton and Ohlsson investigated the distribution of faults within two releases of a system, focusing on four main hypotheses, which were investigated although not formally tested. Firstly, it was shown that some software modules have a greater concentration of faults compared to other modules, i.e. following the Pareto principle [10]. Secondly, they examined the persistence of faults across test phases. One of the hypotheses states that if a higher number of faults in a certain module are detected by function test, a high number of faults will also be detected in that module in system test. From a prediction perspective strong support for this hypothesis would be valuable. However, only weak support for this hypothesis was provided by the case study results. A second perspective on fault persistence in modules concerned whether a high number of faults detected before the system was released, would imply a high number of faults detected after release, in site tests and operation. This hypothesis was strongly rejected. The third area investigated by Fenton and Ohlsson was whether size and complexity of each module have an impact on the number of faults and whether these parameters would function as predictors. The area was divided into several hypotheses, each more thoroughly examining size metrics as predictors for the absolute number of faults and the fault density, whereas the faults were separated into pre-release and post-release faults. The hypothesis concerning complexity compared the prediction ability to the size metrics. None of the hypotheses had any strong support in the case study, only weak support for size metrics as a predictor of the number of pre-release faults was provided. The fourth area investigated fault densities in more detail with a benchmarking strategy. It was shown that the fault densities of corresponding test phases were rather similar for the two releases of the system, and that the fault densities partly were similar to the result of other studies.

Fenton and Ohlsson argue that too few studies, with results in this domain, have been published to be able to do a proper investigation. Here, we contribute by expanding the available empirical knowledge, with results from a study examining three software systems. The software systems are developed by an organization, which characteristics differ widely from the organization studied by Fenton and Ohlsson, and the results will thereby give a new perspective to the hypotheses under investigation.

A similar study is presented by Ostrand and Weyuker, examining fault data from 13 releases of an inventory tracking system [18]. The goal of their study was to find ways to identify fault-prone files, and the analysis was divided into four categories. Firstly, the distribution of faults over the comprising software modules was examined. Here, a strong evidence for a Pareto principle was found, saying that a few of the modules are responsible for a large share of the faults. Secondly, the effect of the modules' size on fault density was examined. The results showed no support for the conventional wisdom that larger modules have higher fault densities. Thirdly, the modules' persistence of faults was examined. Unfortunately, any conclusions on the predictive ability of fault concentration during testing for faults after release could not be gained, due to a low number of detected faults after release. Finally, the fault-proneness of newly written code compared to code written earlier was examined. The analysis confirmed the intuitive "knowledge" that the fault density for new modules was higher than for older ones. The study by Ostrand and Weyuker [18] is extended in a second study, where the original data set of 13 releases is expanded with another 4 releases [19]. The second study focused on predicting the number of faults to expect and their location in the system in each release, based on previous releases.

In addition to the studies by Fenton and Ohlsson, and Ostrand and Weyuker, who take a rather comprehensive view of quantitative issues, other studies investigating fault distributions have been conducted. However, most have investigated only one or a few of the areas in which Fenton and Ohlsson formulated their hypotheses. Two studies in this field were conducted more than twenty years ago [1][3]. Adams' study, consisting of data from nine software products, shows that a small number of faults cause the most common failures [1]. Only when these "most-often-failure-causing" faults are removed, the reliability will be improved significantly. Basili and Perricone [3] analysed fault data from a software project, starting its life cycle in the mid 70's. The investigation focused on comparing fault distributions in modified software modules and newly developed, respectively.

In the 90's a second set of studies are published. Möller and Paulish report on the effects of modification and reuse on fault density [16]. A number of studies have examined the relation between module size and fault density, e.g. [3][16][28]. A review of studies on the subject is conducted by El Emam et al [5]. Hatton has also reviewed a number of studies, focusing on the size of the components and proportionality to

reliability [8]. The results of the studies, including [3][16], imply that decomposing a system into small components would not increase the reliability. The opposite is argued by El Emam et al, who in a study of object-oriented systems show that as class size increase, so will fault proneness [5].

Fenton and Ohlsson found evidence for a subset of the hypotheses they stated [6], but in other cases the opposite of the conventional wisdom was shown. The empirical knowledge is further built by the studies mentioned above, but still several of the areas investigated in the case study by Fenton and Ohlsson need more examination and analysis. Further empirical research in the field and quantitative analysis of fault data is required to support or disprove the conjectures given in these studies. In addition, a few of the results even conflicts with conventional wisdom and sustain the argument of more research in the field.

3. Hypotheses

Fenton and Ohlsson listed a number of hypotheses and analysed empirically whether the hypotheses could be supported or rejected. Four categories of questions were addressed. The same categories are listed below, each with formulated hypotheses. The hypotheses are kept the same in the replication study to provide results that can be combined and compared to the results from the original study, to build up knowledge about software fault distributions.

- Evidence of the Pareto principle of distribution of faults: Hypotheses relating to how faults are distributed over different modules, and the conventional wisdom that a small percentage of modules will contain a large percentage of faults detected.

  1a. A small number of modules contain most of the faults detected during pre-release testing.

  1b. If a small number of modules contain most of the pre-release faults, then this is because those modules constitute most of the code size.

  2a. A small number of modules contain most of the faults detected during post-release testing.

2b. If a small number of modules contain most of the post-release faults, then this is because those modules constitute most of the code size.

- Persistence of faults: Hypotheses relating to that modules with high fault concentration in the early test activities will tend to have high fault concentration in later test activities.

  3. A higher incidence of faults in function testing implies higher incidence of faults in system testing.

  4. A higher incidence of faults in pre-release testing implies higher incidence of faults in post-release testing and use.

- Effects of module size and complexity on fault-proneness: Hypotheses about metrics potentially suitable for fault prediction.

  5. Simple size metrics, such as Lines of Code (LOC), are good predictors of fault and failure prone modules.

  6. Complexity metrics are better predictors than simple size metrics of fault and failure-prone modules. This hypothesis is *not* included in the analysis in the replication study.

- Quality in terms of fault densities: Hypotheses relating to fault densities separated for each test activity, and pre- and post-release testing.

  7. Fault densities at corresponding phases of testing and operation remain roughly constant between subsequent major releases of a software system.

  8. Software systems produced in similar environments have broadly similar fault densities at similar testing and operational phases (benchmarking figures from other studies).

We have addressed these hypotheses, and analysed them using new empirical data. The data originate from three unique software systems, and not as in the original study by Fenton and Ohlsson successive releases of the same system. However, the systems investigated in the replication study relate to each other in a product-line context, and have thus some common components.

Our analyses of the hypotheses differ slightly from the original study. We have applied statistical data analysis techniques, where the original study mostly relied on descriptive statistics. Hypothesis 6, concerning

whether complexity metrics are useful as predictors of fault-proneness, is not discussed in this study, as we did not have access to data on the complexity of each module. Hypothesis 7 concerns fault densities of subsequent releases of a system. As we have one single release of each system, the hypothesis is not precisely comparable between the studies. However, the developed systems are similar and it is our opinion that the hypothesis concerning fault density of software systems developed subsequently is of as much interest.

3.1 Data analysis

The primary method used for data analysis is graphical techniques. Scatter plots are used to show the variation between particular variables for the included modules. The Alberg diagram [17], suggested by Fenton and Ohlsson, is used to order the modules with respect to the variable of interest. In addition to the graphical technique used by Fenton and Ohlsson for investigating the hypotheses, we have applied statistical data analysis techniques. To test whether two sets of data are related, and if so the degree of their relation, a measure of association is calculated. The measure of correlation used is the Pearson product-moment correlation coefficient [7]. This correlation coefficient measures only relationship of the type linear dependency.

Even though statistical analysis is performed, generality is of course not a matter. The objective of statistical interference is to draw conclusions about a population from which we have made a sample. When the wish is to make a general statement, defining the population is a problem. Furthermore, to make general statements we have to be confident to have done a random sampling from the population, in this case a series of software projects. The collected data originate from a single organization; thereby the results are generalized to that organization, but not to a wider community. Hence, the results have to be combined with findings from other studies to give an overview and provide practical applicability to other environments.

4. System description

The hypotheses are investigated using empirical data from a large company in the telecommunications domain. The company develops

handheld devices with a substantial proportion of its functionality implemented in software. Data from three separate software development projects are analysed. Two of the projects are application projects, developing end user products, whereas one project was an internal platform project, where the final product is used as a platform in other end user products. We refer to the projects as *Project 1*, *Project 2* and *Project 3*, respectively. *Project 1* and *3* are application projects, resulting in the complete software for two unique end user products. *Project 2* is the platform project. From the research purpose, much of the detailed data of the projects are not of interest, and instead of absolute figures relative comparisons to the original study by Fenton and Ohlsson are presented. A subset of modules in each project is included in the analysis, 45, 90 and 90 modules respectively from the three projects. The subset is chosen to include modules from each existing group of functionality. Similar to Fenton and Ohlsson, we are not permitted to present the total size of the respective projects, but the code base in the products is counted in millions of lines of code. However, comparing the largest modules, the modules in this study are more than 10 times larger than in the original study, as shown in Table 1.

Table 1. Distribution of modules by size.

LOC	Project 1	Project 2	Project 3
<10 000	12	30	25
10 001-30 000	14	28	29
30 001-50 000	8	14	16
50 001-70 000	6	6	3
70 001-90 000	1	3	4
>90 000	4	9	13
Total	45	90	90

As mentioned in Section 3, the complexity measure of each module is not available, and thereby excluded from the analysis.

A summary of project characteristics is given in Table 2. More information on the data collection process could be found in [2].

As in the original study the dependent variable is the number of faults. Each fault is located in a module, i.e. if the cause of a detected failure was corrected by modifying *n* modules, it is counted as *n* distinct faults. Even if the specific correction in a module concerns several lines of code, this

Table 2. Project characteristics.

	Project type	Sample size (# modules)	System size (scaled)	# Faults (scaled)
Project 1	Application	45	10	10
Project 2	Platform	90	15	25
Project 3	Application	90	20	15

counts as one fault. Only faults regarding the specific product, or in the case of Project 2 the platform, were included. Change proposals are filed as change requests and not as failures, hence these were excluded from the analysis. Duplicate failures, which are detected more than once, were also excluded.

The total number of faults is in the same range as in the original study. In the original study, *release n* contained 1669 faults and *release n+1* contained 3646 faults. We are not permitted to specify the absolute figures, but the percentage distributions of the number of faults detected by each test activity are given in Table 3, together with the percentage distributions from the original study. In Table 3 it is shown that at least for *release n+1* the percentage distribution of faults are rather similar to the distributions investigated in the replication study. In the replication study data are collected during function test, system test, customer acceptance test and the first couple of months of operation. Faults detected during function test and system test are as in the original study grouped into pre-release faults, whereas faults detected during customer acceptance test and first months of operation are combined into post-release faults. This latter group of faults is not separable in this study, as in the study by Fenton and Ohlsson, who combined faults from the "first 26 weeks of site tests" and "approximately the first year of operation after the site tests".

Table 3. Percentage distributions of faults per testing activity.

		Pre-release faults		Post-release faults
		Function test	System test	Site and operation
Original study	Release n	55%	41%	4%
	Release n+1	63%	28%	9%
Replication study	Project 1	72%	23%	5%
	Project 2	67%	29%	4%
	Project 3	62%	28%	10%

5. Data analysis and results

In this section we analyse each hypothesis, and compare the results to related work. Primarily the original study by Fenton and Ohlsson is used for comparison, but also other studies are used when appropriate.

5.1 Hypothesis 1: Pareto principle regarding pre-release faults

The first category of hypotheses deals with the belief that a small number of modules in a system contain a substantial part of the existing faults. The Pareto principle was originally formulated by the Italian economist Vilfredo Pareto [23], but later generalized by Juran [10], suggesting that most of the results in any context are determined by a small number of causes. The principle has been applied in various contexts, and is often referred to as the 20-80 rule, in this context suggesting that 20 percent of the modules contain 80 percent of the faults. We investigate however the general principle and do not hypothesize any specific percentage distribution.

Four hypotheses concerning the Pareto principle have been examined in this study, two of them concerning pre-release faults, which are described below:

Hypothesis 1a: A small number of modules contain the faults that cause most of the failures discovered during pre-release testing.

The first hypothesis focuses on pre-release faults, i.e. faults detected during function test and system test. Figure 1 shows the percentage of accumulated number of faults, when the modules are ordered with respect to the number of faults in each. In Figure 1 the three projects can be seen, indicating that 20 percent of the modules in Project 1 are responsible for 63 percent of the faults discovered during pre-release testing. For Project 2, 20 percent of the modules are responsible for 70 percent of the faults discovered during pre-release testing, while Project 3 has an almost identical result, 20 percent of the modules are responsible for 70 percent of the faults discovered during pre-release.

This result is somewhat stronger than reported in the original study (20-60) [6], as well as by Kaaniche and Kanoun (38-80) [11] and Munson and Khoshgoftaar (20-65) [15]. Still, there are studies showing

even stronger focus on fewer modules. Compton and Withrow found a 12-75 relation in their study [4]. Hence, our results are within the range of what is observed before, and it rather confirms the hypothesis than rejects it.

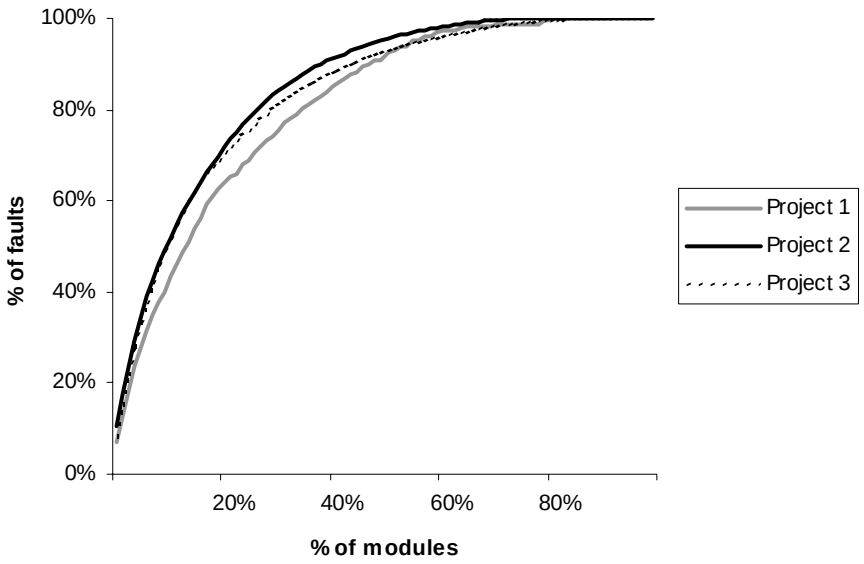

Figure 1. *Alberg diagram showing percentage of modules versus percentage of pre-release faults for the projects.*

Hypothesis 1b: If a small number of modules contain most of the faults discovered during pre-release testing, then this is simply because those modules constitute most of the code size.

Similar to the discussion in the original study, strong evidence for Hypothesis 1a might be explained by the statement that those modules containing a majority of the faults constitute a large amount of the total system size. So was the case in the Compton and Withrow study, where the 12 percent of modules accounted for 63 percent of the lines of code [4].

Similar to the results in the original study, we found no evidence to support the hypothesis, see Table 4. For Project 1, those 20 percent of the modules responsible for 63 percent of the faults made up 38 percent of the code, while for Project 2, those 20 percent of the modules responsible

for 70 percent of the faults made up 25 percent of the code. For the last project, Project 3, the 20 percent of the modules responsible for 70 percent of the faults made up 39 percent of the code.

The results as such for hypothesis 1a and 1b show that whereas we have a larger proportion of faults in a smaller portion of the code base, the difference is not that large as when comparing a smaller proportion of the number of modules to the number of faults. From a practical perspective the first hypothesis still has more importance, indicating that we have a set of modules, which are more fault-prone than the rest.

Table 4. Percentage distributions of pre-release faults over components, related to size.

		Share of modules	Share of pre-release faults	Share of system size
Original study	Release n	20%	60%	30%
	Release n+1	"almost identical to release n"		
Replication study	Project 1	20%	63%	38%
	Project 2	20%	70%	25%
	Project 3	20%	70%	39%

5.2 Hypothesis 2: Pareto principle regarding post-release faults

Hypothesis 2 concerns post-release faults, i.e. faults detected in customer acceptance test and first months of operation, suggesting that the Pareto principle holds also for these faults.

Hypothesis 2a: A small number of modules contain the faults that cause the most operational failures.

Figure 2 illustrates for the three projects the percentage of accumulated number of post-release faults, when the modules are sorted in decreasing order with respect to fault content. For Project 1, it could be seen that 20 percent of the modules are responsible for 87 percent of the post-release detected faults, whereas the result for Project 2 is almost identical. For Project 3, 20 percent of the modules are responsible for 80 percent of the post-release faults. Compared to Hypothesis 1a, we have an even stronger support for the hypothesis, having a large amount of the post-release faults in a small proportion of the modules.

In the original study, Fenton and Ohlsson examined how large proportion of post-release faults 10 percent of the modules are responsible for. For our study, 10 percent of the modules in Project 1 are responsible for 63 percent of the post-release faults, while the same amount of modules for Project 2 are responsible for 74 percent of the post-release faults, and for Project 3, 59 percent of the post-release faults, see Table 5.

Table 5. Percentage distributions of post-release faults over components.

		Share of modules	**Share of faults**
Original study	Release n	10%	100%
	Release n+1	10%	80%
Replication study	Project 1	10%	63%
	Project 2	10%	74%
	Project 3	10%	59%

The original study provided evidence for this hypothesis. 10 percent of the modules are responsible for 100 and 80 percent of the failures found, for the two releases respectively. In our study, the post-release faults are more scattered over the system. It is however worth noting that the concept of post-release faults is weaker in the replication study than in the original study. In the replication, customer acceptance test faults and only the first months of operation belong to this category, while in the original study post-release faults include one year of operation.

Hypothesis 2b: If a small number of modules contain most of the operational faults, then this is simply because those modules constitute most of the code size.

Since we found evidence for Hypothesis 2a, Hypothesis 2b is justified to investigate. As in Hypothesis 1b, the support of Hypothesis 2a could be explained by the statement that those modules containing a majority of the post-release faults constitute a large amount of the total system size. However, we did not find evidence for this. For Project 1, those 20 percent of the modules responsible for 87 percent of the faults made up 24 percent of the code, while for Project 2, those 20 percent of the modules responsible for 88 percent of the faults made up 22 percent of the code. For Project 3, those 20 percent of the modules responsible for 80 percent of the faults made up 40 percent of the code. Thus, we found no support for the hypothesis in our study.

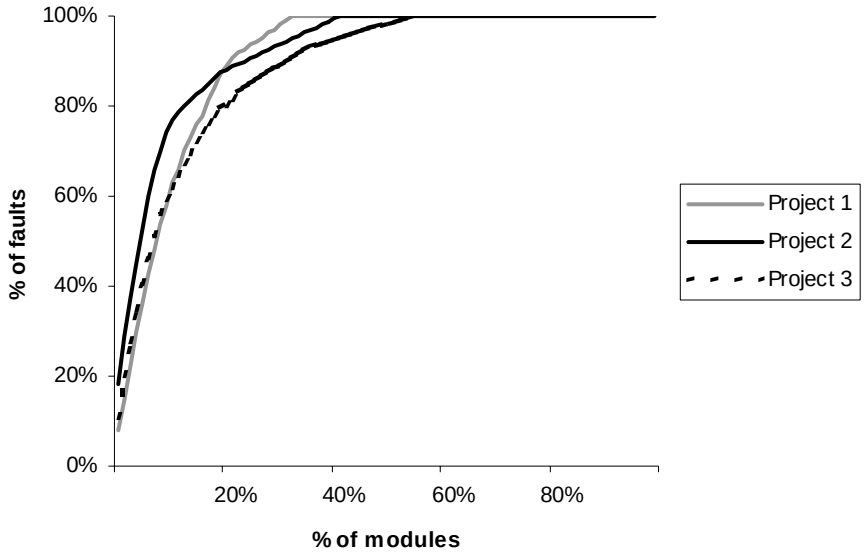

Figure 2. *Alberg diagram showing percentage of modules versus percentage of post-release faults for the projects.*

In contrast to the original study, we did not find strong evidence in favour of the converse hypothesis either. For Project 1, 100 percent of the post-release faults were contained in modules that make up 32 percent of the system. For Project 2, 100 percent of the post-release faults were contained in modules that make up 41 percent of the system, while for Project 3, 100 percent of the post-release faults were contained in modules that make up 70 percent of the system. In summary, the original study showed strong evidence of the converse hypothesis, whereas the replication study did not, see Table 6.

Table 6. Percentage distributions of post-release faults, related to size.

		Share of post-release faults	**Share of system size**
Original study	Release n	100%	12%
	Release n+1	78%	10%
Replication study	Project 1	100%	32%
	Project 2	100%	41%
	Project 3	100%	70%

5.3 Hypothesis 3: High incidence of faults in FT implies high incidence of faults in ST

A common belief is that modules with a high fault incidence in the early phases of testing will imply a high fault incidence in later phases of testing as well. Thus, the hypothesis suggests that a high detection rate of faults in function test in a particular module will result in a high detection rate of faults in system test in the same module.

Hypothesis 3: A higher incidence of faults in FT implies a higher incidence of faults in ST.

The hypothesis is evaluated from different perspectives. Figure 3 illustrates the relation between the number of faults detected in function test and system test. Each dot represents a module. Statistical analysis shows that we have a correlation between the number of faults in function test and system test. The Pearson correlation coefficient r is used as measure of association. For project 1: $r_1 = 0.74$, for Project 2: $r_2 = 0.84$, and for Project 3: $r_3 = 0.68$. The p-values for each of the correlation coefficients are small, $p_i < 0.001$, $i = 1, 2, 3$.

The correlation between number of faults detected in function test and number of faults detected in system test indicates that the most fault-prone modules in the early phase of testing will be fault-prone also when tested in system test. In the analysis the modules are ordered with respect to the number of faults in system test, and show that for Project 1: 50 percent of the faults detected in system test occurred in modules, which

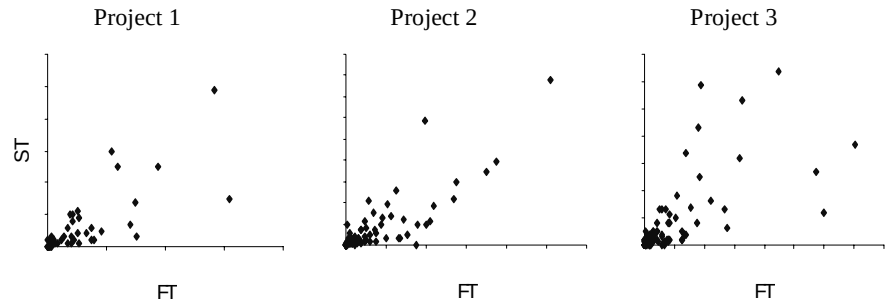

Figure 3. *Scatter plot showing relationship between faults detected in function test and faults detected in system test.*

were responsible for 40 percent of the faults detected in function test. For Project 2: 50 percent of the faults detected in system test occurred in modules, which were responsible for 39 percent of the faults detected in function test, while for Project 3: 50 percent of the faults detected in system test occurred in modules, which were responsible for 38 percent of the faults detected in function test. Figure 4 show that for each project these 50% of the system test faults are contained in a small number of modules. The corresponding numbers in the original study are that 50 percent of the faults detected in system test occurred in modules, which were responsible for 37 and 25 percent of the faults detected in function test, respectively for *release n* and *n+1*.

This information can be used to predict fault contents over modules, see Figure 4. The figure shows for Project 1 that 10 percent of the most fault-prone modules in system test are responsible for 53 percent of the faults in system test (solid line), whereas (when the modules are ordered with respect to fault-proneness in function test) the 10 percent of the most fault-prone modules in function test are responsible for 39 percent of the faults in system test (dashed line). The results for Project 3 are similar, see Table 7, while for Project 2 the results differ some. For Project 2 the 10 percent most fault-prone modules in function test are responsible for 52 percent of the faults in system test, while the 10 percent most fault-prone modules in system test are responsible for 56 percent of the faults in system test, which leaves almost no faults left to be explained. The results could be expected from the correlation analysis, where specifically Project 2 had a high correlation between the number of faults in function test and the number of faults in system test. The systematic difference we can see between the three projects is that Project 2 is a platform project, while projects 1 and 3 are application projects.

Table 7. Faults in system test for the 10 percent most fault-prone modules, when ordered with respect to fault-proneness in system test and function test, respectively.

		ST	**FT**
Original study	Release n	38%	17%
	Release n+1	46%	24%
Replication study	Project 1	53%	39%
	Project 2	56%	52%
	Project 3	56%	39%

The original study did not present any correlation analysis for this hypothesis. Further, the proportions were less coherent compared to our study, hence not concluding any strong results from the data. We interpret on the contrary the replication data as showing support for the hypothesis.

5.4 Hypothesis 4: High incidence of pre-release faults implies high incidence of post-release faults

Similar to Hypothesis 3, the belief that modules, which are fault-prone in early phases of testing will be fault-prone in later phases as well, works as a rationale for the fourth hypothesis. It is assumed that if a module has had a high incidence of faults before released to the customer, the module will have a high incidence of faults after release as well.

Hypothesis 4: A higher incidence of faults in pre-release testing implies higher incidence of failures in operation.

Figure 5 illustrates the relationship between the number of pre-release faults and the number post-release faults. Each dot represents a module. Statistical analysis gives that there is a moderate correlation between the two parameters. The measure of association for each project is calculated by Pearson correlation coefficient *r*. For Project 1 is $r_1 = 0.67$, for Project 2: $r_2 = 0.56$, and Project 3: $r_3 = 0.72$. The p-values for each of the correlation coefficients are small, $p_i < 0.001$, $i = 1, 2, 3$. In contrast to Fenton and Ohlsson, who even found support for the converse hypothesis, our results give some support the hypothesis.

Fenton and Ohlsson showed that almost all of the faults detected in pre-release testing appear in modules that subsequently had no post-release faults. 93% and 77% of the faults in pre-release testing occurred in modules that had no post-release faults for the two releases respectively. Unlike their results, we observed that in Project 1: 36 percent of the faults in pre-release testing occur in modules that have no subsequent post-release faults. For Project 2: 29 percent of the faults in pre-release testing occur in modules which have no subsequent post-release faults, while for Project 3 only 13 percent of faults in pre-release testing occur in modules which have no subsequent post-release faults.

Based on this data, the original study rejects the hypothesis, and makes a note that this is a remarkable result. Compton and Withrow [4] however, found much more post-release fault in modules that also had

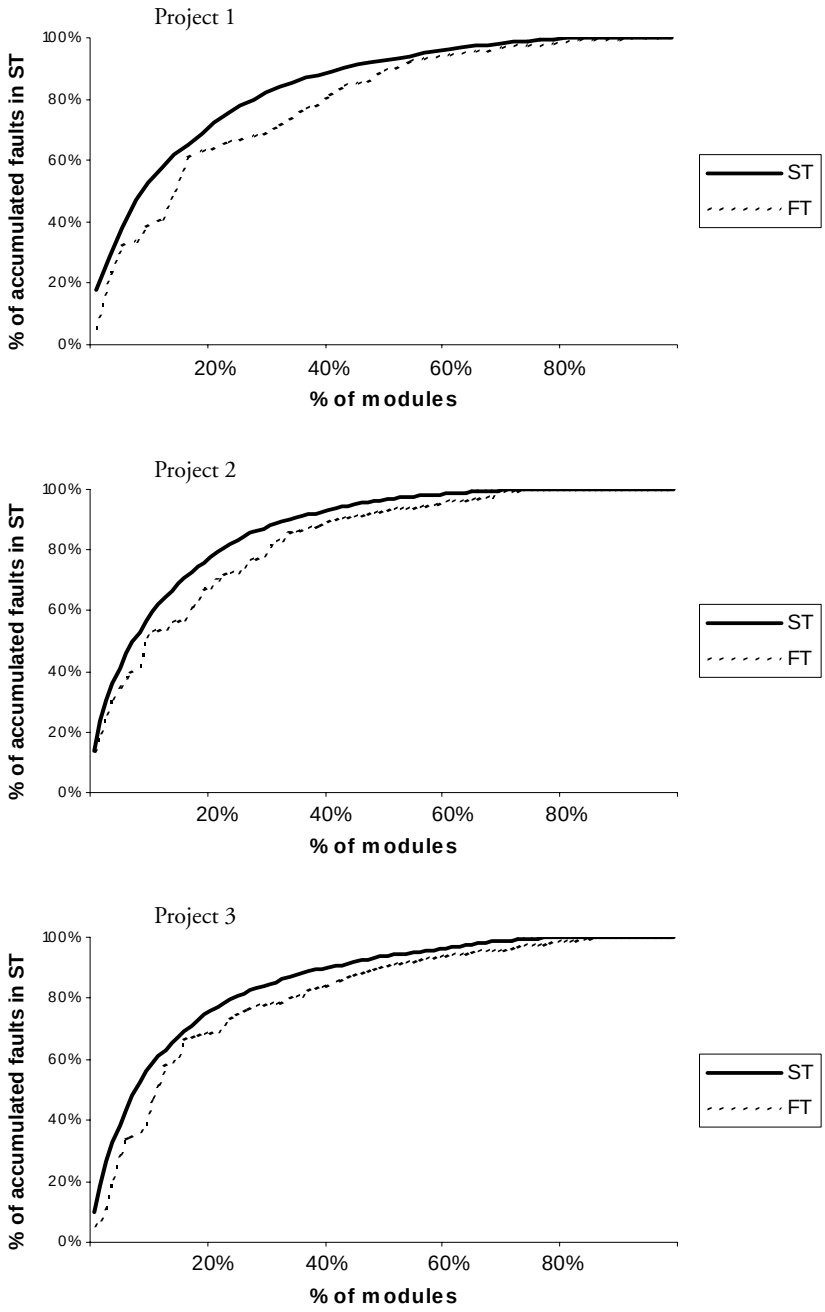

Figure 4. *Accumulated percentage of the number of faults in system test when modules are ordered with respect to the number of faults in system test versus function test.*

pre-release faults. It is clear from the contradictory results, that there are factors at work, which are not measured in these case studies. For example, if the pre-release testing is thorough for certain modules, one can expect that they will not contribute to post-release faults, although if the pre-release testing is not sufficient, that will lead to finding post-release faults for those components.

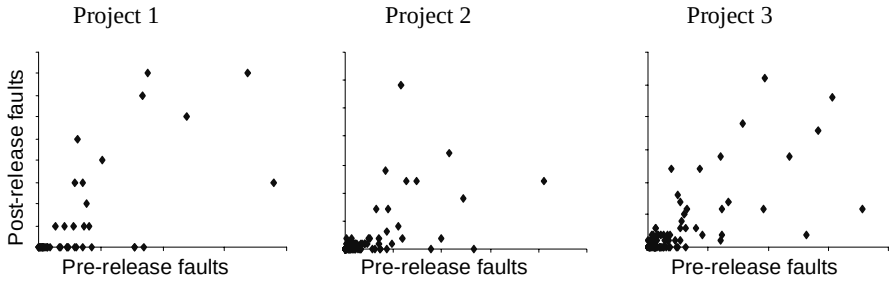

Figure 5. *Scatter plot showing relationship between pre-release faults and post-release faults.*

5.5 Hypothesis 5: Size metrics as fault predictors

In this section we focus on hypotheses about size metrics for predicting which modules will be fault-prone. The analysis of the effect of module size is separated into five different hypotheses, closely related, but still giving different aspects of module size effect.

Hypothesis 5a: Smaller modules are less likely to be failure prone than larger ones.

A number of studies have investigated the relationship between module size and the number of faults, and produced counter-intuitive results. The conventional wisdom says that smaller modules should result in fewer faults, since small modules are easier to develop. On the other hand, a large amount of small modules will result in more interface faults, spread across the modules.

We have analysed Hypothesis 5a both in terms of the total size of the modules in LOC, and in terms of the absolute value of the new LOC.

Table 8. For each hypotheses, correlation between module size and number of faults vs. fault-density

	5a. Total number of faults		5b. Pre-release faults	5c. Post-release faults	5d. Pre-release fault-density	5e. Post-release fault-density
	Total LOC	New LOC				
Project 1	0.38	0.05	0.37	0.44	-0.17	-0.15
Project 2	0.05	0.04	0.05	0.08	-0.22	-0.16
Project 3	0.62	0.32	0.60	0.65	-0.10	-0.02

New LOC could be added code, changed code, and deleted. Column 2 and 3 in Table 8 gives the measures of association for the investigated projects.

In the original study, Hypothesis 5a was investigated by showing the number of modules that had a certain number of faults [6]. Similar to the study by Basili and Perricone [3], a table is used, which groups modules according to the frequency of faults found. Compared to the results in these studies, the data sets in the replication study have a lower proportion of modules with few faults. While Fenton and Ohlsson did not find support for the hypothesis, we have found some support, although it is not consistent for all the included projects. There is some correlation between the total number of faults and the total lines of code for Project 3, whereas for Project 1 and Project 2 the correlation coefficients are low. For Project 3, the *p-value* for the correlation coefficient between total number of faults and total LOC is < 0.001 and is showing on statistical significance. The other correlation coefficients are not indicating this. The measure of association is less when calculated for new lines of code compared to the total number of faults. In the following analysis, the hypotheses are investigated with the total lines of code of each module.

Hypothesis 5b: Size metrics are good predictors of pre-release faults in a module.

Figure 6 illustrates the relationship between module size in total LOC and the number of faults detected in pre-release testing for the projects. The correlation of the relationship is given in column 4 in Table 8, showing a very low value for Project 2. On the other hand, the measure of association for Project 3 gives moderate support for the hypothesis, with a *p-value* < 0.001.

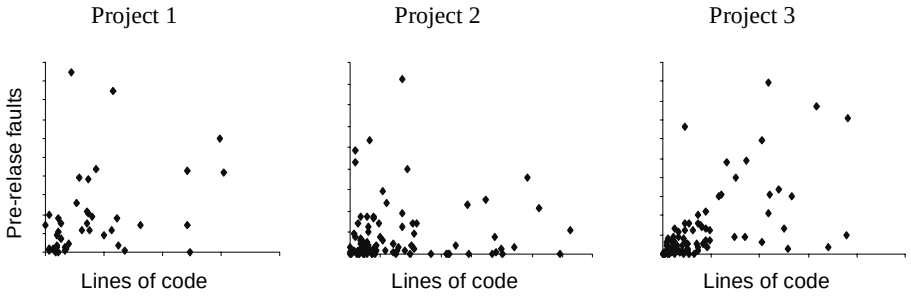

Figure 6. *Scatter plot showing relationship between LOC and pre-release faults.*

This is the single hypothesis, related to size metrics, where Fenton and Ohlsson observed some support in their scatter plots, although only weak support.

Hypothesis 5c: Size metrics are good predictors of post-release fault in a module.

Figure 7 illustrates the relationship between module size in total LOC and the number of faults detected in post-release testing for the projects. The correlation of the relationship is given in column 5 in Table 8. Similar to Hypothesis 5b, the measure of association for Project 2 is very low, giving no support for the hypothesis, whereas the measure of association for at least Project 3 gives some support for the hypothesis. The *p-value* for the correlation coefficient for Project 1 is 0.003, and for Project 3: < 0.001.

In the original study, Fenton and Ohlsson did not find any support for the hypothesis.

Hypothesis 5d: Size metrics are good predictors of a module's (pre-release) fault-density.

Studying fault density and module size could easily be misleading. The relationship between a variable *X* and *1/X* will always be negative, which thereby is the case between size and fault density [26]. The relationships between size and the number of faults, which were analysed in hypotheses 5b and 5c, are more relevant, but from the replication perspective we present the analysis of hypotheses 5d and 5e. Hence, the hypotheses are also investigated with data normalized to the number of lines of code

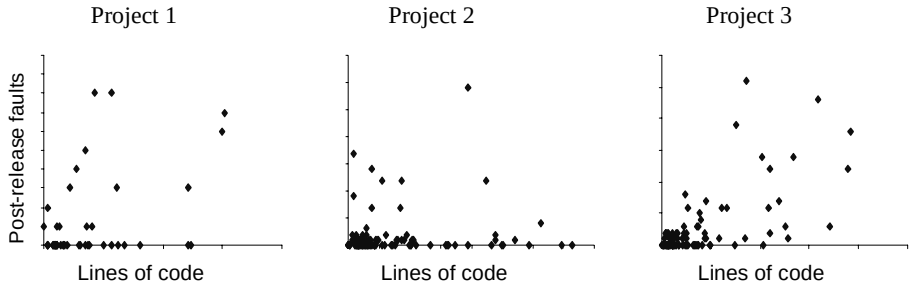

Figure 7. *Scatter plot showing relationship between LOC and post-release faults.*

instead of absolute values, investigating the fault density of the modules, although there is a major methodological threat in the analysis.

Figure 8 illustrates the relationship between total LOC and pre-release fault density of each module for the projects. The correlation between the parameters is given in column 6 in Table 8. The correlation is as expected negative, suggesting an inverse relationship between fault density and lines of code, i.e. we have higher fault-densities for smaller modules. However, the measures of association for the projects have low values, and do not give any support for the hypothesis.

In the original study, when examining the fault density to module size, Fenton and Ohlsson did not find any trend at all. The results differ from several other studies, where a relation between fault density and module size is shown [8][16][18]. On the other hand, Fenton and Ohlsson argue

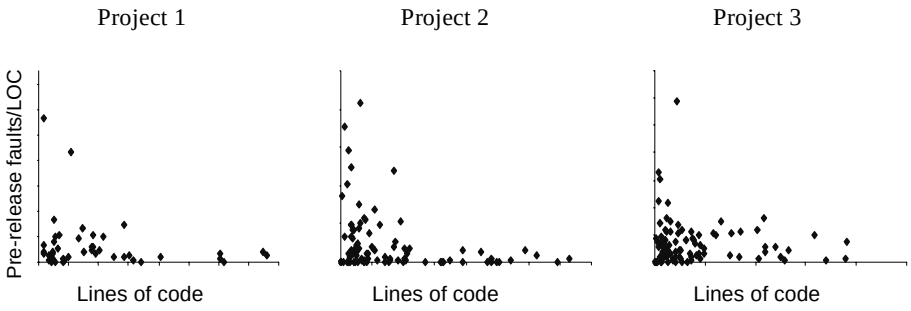

Figure 8. *Scatter plot showing relationship between pre-release fault density and size.*

that the analysis conducted in some studies is misleading because of the grouping of data. By analysing simple scatter plots instead of comparing average figures for different size intervals, Fenton and Ohlsson do not receive any evidence for a relationship between fault density and module size. They considered their results as a confirmation that there is no causal relationship between size and fault density.

Hypothesis 5e: Size metrics are good predictors of a module's (post-release) fault-density.

Similar to Hypothesis 5d, the relationship between module size and fault density is investigated, now post-release faults instead of pre-release faults. Figure 9 illustrates scatter plots for lines of code versus post-release fault density for the projects. The correlation is given in column 7 in Table 8. Similar to the results from the previous hypothesis, the measures of association are negative, suggesting that smaller modules have higher fault density for post-release faults. However, the values are very low, and do not give any support for the hypothesis, and the methodological threat is inherent in this analysis as well.

In the original study, no support for the hypothesis was found either.

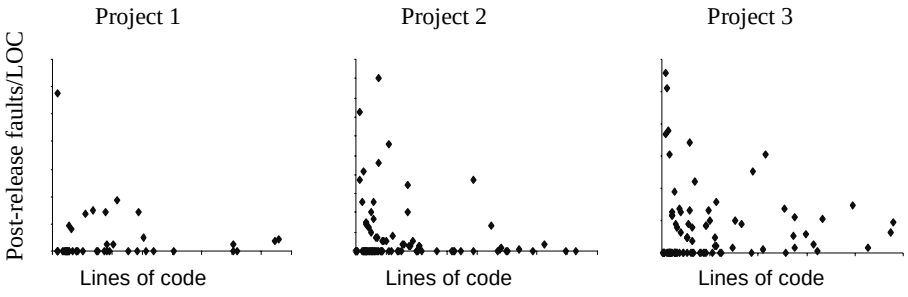

Figure 9. *Scatter plot showing relationship between module post-release fault density and size.*

In addition to the hypotheses investigated above, concerning size metrics and fault-proneness, the ranking ability of lines of code is assessed. Fenton and Ohlsson concluded that the lines of code metric worked rather well at predicting the most fault-prone modules, despite that the analysis did not give any evidence to support the hypotheses. Our

results do not unambiguously indicate that LOC works as predictor for ranking the most fault-prone modules. The Alberg diagrams in Figure 10 illustrate the LOC ranking ability for the projects. For Project 1, 20 percent of the largest modules are responsible for 26 percent of all faults, while for Project 2, 20 percent of the largest modules are responsible for 18 percent of all faults. These two projects do not indicate any good ranking ability for lines of code. However, for Project 3, 20 percent of the largest modules are responsible for 57 percent of all faults; a large amount of the detected faults. The diagrams for the two former projects do not reveal any good predictability, and LOC works less well than in the original study by Fenton and Ohlsson at ranking the most fault-prone modules. On the other hand, for the third project a few of the modules are evidently more fault-prone than the rest, and indicate a better ranking ability on fault-proneness than the results from the original study.

Summarizing the analysis of size metrics as predictor for fault-proneness, the results are varying. The results of each project diverge from each other, although the development environment is the same and other affecting parameters agree. However, it is clear that some parameters are not the same. For example, Project 2 is a large platform project, not resulting in a final end-product as the application projects. For Project 2 a large proportion of the code is newly developed compared to Project 1 and 3 where more code is reused instead.

5.6 Hypothesis 7: Fault-densities at corresponding test phases

The purpose of hypotheses 7 and 8 is to visualize the range of fault densities that could be expected in software development and testing, and to continue to reach a consensus about what fault density could be expected under certain circumstances considering e.g. development environment, testing activities and programming languages.

We have not investigated the hypotheses on subsequent releases as in the original study, but on the three subsequent projects, each delivering a unique system. However, we considered the hypotheses to be interesting to evaluate, since the projects are executed in the same environment, close in time to each other, and they share software components, although the ratio of common code is unknown.

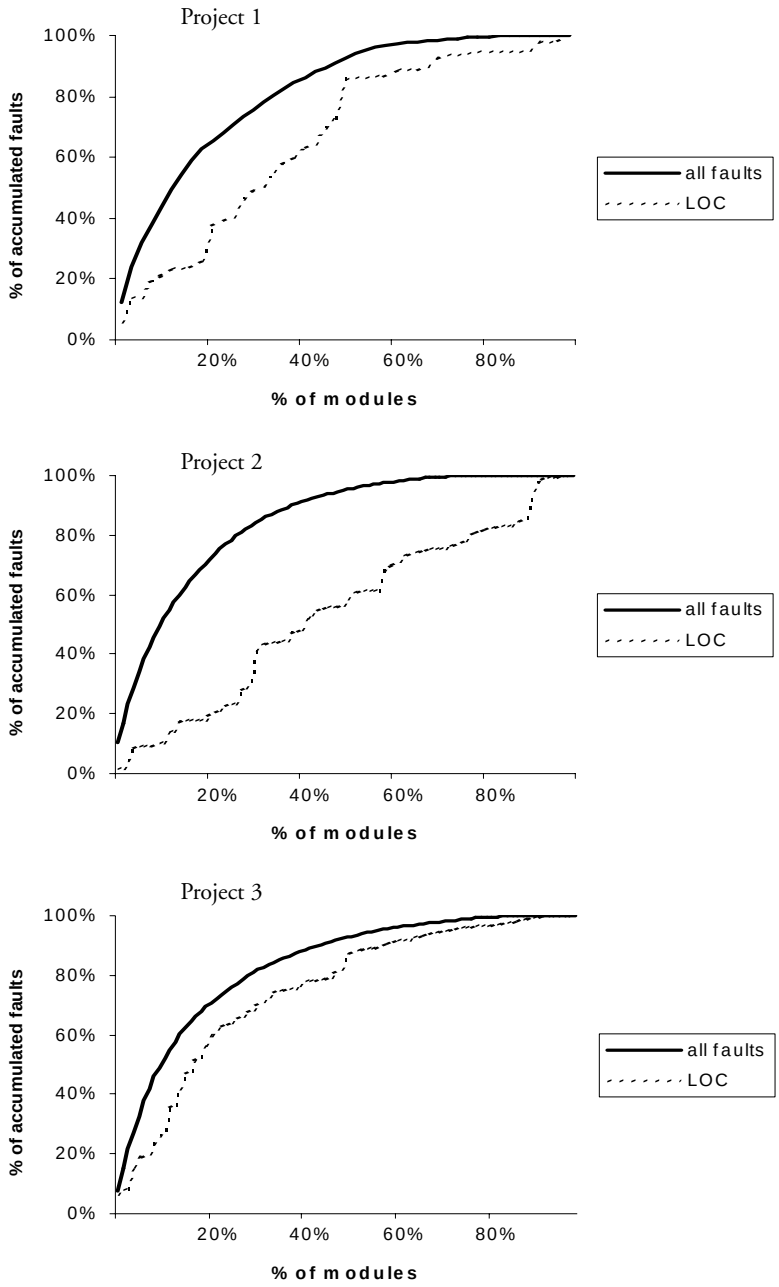

Figure 10. *Accumulated percentage of number of faults when modules are ordered with respect to LOC.*

Hypothesis 7: Fault densities at corresponding phases of testing and operation remain roughly constant between subsequent projects, conducted in the same context.

The results from this study show that the fault densities are in the same magnitude of order for the same test activities in the three projects, see Table 9. Project 2 has higher fault densities for the function test and the system test activities, while Project 1 and 3 have similar fault densities for these activities. The results of the rather similar fault densities show that the development process used in the investigated organization is stable and repeatable.

Nevertheless, the fault densities for the test activities for Project 2, which differs from the other projects, could be analysed further. The project as such is larger than Project 1, but smaller than Project 3, in terms of total lines of code. However, the proportion of newly developed code is larger in Project 2 than in the other two projects. This could have resulted in a higher number of injected faults, which lead to that the testers had more faults to detect.

As shown in Section 4 and Table 3, the test activities have similar distributions for fault detection, regardless of the other project specific characteristics like size, duration and number of detected faults. Although test effort, evidently an important parameter in this subject, is excluded from the analysis, the distributions in Table 3 present a view of test activities' detection capacities, also suggesting that the development process is stable.

Table 9. Fault densities for the testing activities (faults/kLOC).

	FT	ST	CAT
Project 1	0.7	0.2	0.03
Project 2	1.2	0.5	0.07
Project 3	0.4	0.2	0.07

5.7 Hypothesis 8: Fault-densities of systems from similar environment

Hypothesis 8: Software systems produced in similar environment have broadly similar fault densities at similar testing and operation phases.

In the original study, for Hypothesis 8 Fenton and Ohlsson compared their observed fault densities for pre- and post release activities with other published data, suggesting that "software systems developed in similar environment have broadly similar fault densities at similar testing and operational phases". We add to that analysis, the data for our three projects, shown in Table 10. Note that the absolute figures are not relevant to compare, as different programming languages are used. When comparing the result of the three projects in the case study, we can see that the pre-release fault densities are in an order of magnitude higher than post-release densities, similar to the results in the original study. Although, the potential confirmation of the hypothesis depends on what is meant by "similar". Among the three studied projects, we identify larger dissimilarities between the application projects and the platform project, while the two application projects are closer. Thus, to some extent the general conclusions drawn in the original study and related work [20] is supported, by showing a higher ratio of pre-release fault density to post-release.

Table 10. Fault densities pre-and post-release (faults/kLOC).

		Pre-release	**Post-release**	**All**
Original study	Release n	6.09	0.27	6.36
	Release n+1	5.97	0.63	6.60
Replication study	Project 1	0.9	0.03	0.93
	Project 2	1.7	0.07	1.77
	Project 3	0.6	0.07	0.67

5.8 Summary

In this section we summarize and interpret the results of the hypotheses under study. The results are reported in Table 11 and commented below, grouped according to the hypotheses on *Pareto principle*, *Faults early vs. late*, *Faults vs. size* and *Fault densities across releases*.

Confirmed hypotheses, where correlations between two metric values are found, do not imply causal relationship. The results as such do not explain the observed behaviour. For example, where a high number of faults are detected, could simply be because these modules have been tested more thoroughly, and not be dependent on the various metrics included in the analysis.

Table 11. Summary of hypotheses in the original study [6] (Table 7), this replication and other empirical studies.

	Hypothesis	Original study	Replication	Other
Pareto principle	1a. Few modules contain most faults (pre-release)	Confirmed (20-60)	Confirmed (20-63; 20-70; 20-70)	(20-65) [15]; (38-80) [11]; (12-75) [4]
	1b. Few faulty modules constitute most of the size (pre-release)	No support (20-30)	No support (20-38; 20-25; 20-39)	(12-63) [4]
	2a. Few modules contain most faults (post-release)	Confirmed (10-80; 10-100)	Confirmed (10-63; 10-74; 10-59)	(38-54) [11]
	2b. Few faulty modules constitute most of the size (post-release)	No support; strong evidence of a converse hypothesis (100-12; 60-6)	No support (100-32; 100-41; 100-70)	
Faults early-late	3. High fault incidence in FT implies the same in ST	Weak support (50-25)	Strong support (50-40; 50-39; 50-38)	
	4. High fault incidence in pre-release implies the same in post-release	No support; strongly rejected (93-0; 77-0)	Support (36-0: 29-0; 13-0)	Support [4]
Faults vs. size	5a. LOC is a good predictor of faults	No support	Varying support	Support converse hypothesis [3]
	5b. LOC is a good predictor of pre-release faults	Weak support	Varying support	
	5c. LOC is a good predictor of post-release faults	No support	Varying support	
	5d. LOC is a good predictor of pre-release fault density	No support	No support	Confirmed [8][16][18]
	5e. LOC is a good predictor of post-release fault density	No support	No support	
Complexity	6. Complexity metrics are good predictors of faults	No (for cyclomatic complexity), some weak support for specific metric	N/A	
Fault densities	7. Fault densities are constant between releases	Confirmed	Confirmed	
	8. Fault densities are similar in similar environments	Confirmed	Confirmed	Confirmed [20]

The results regarding the Pareto principle for fault distributions over modules are to a large extent the same in the replicated study as in the original study. A small set of modules contains a majority of the faults, both regarding pre- and post-release faults. This is in line with other empirical studies [4][11][15][18], although the exact proportions are different from study to study. This result is not due to that these few modules are larger than other modules. The original study found support for the converse hypothesis, i.e. small modules are more fault prone, but this is not the case in our study. Hence, the replication gives additional support for that the Pareto principle is valid in fault distribution over modules.

Another view of investigating the assumption that a few modules contain a majority of the faults is studied in terms of fault contents in modules in early and late phases respectively. The hypotheses state that high fault incidence in a module in function test implies the same in system test, and high pre-release fault incidence imply high post-release fault incidence. The original study found weak support for the first hypothesis, and no support for the latter. In the replication study, we found support for both hypotheses. The original report comments on their results as being "remarkable" [6]. We tend to consider the results of the replication study being more intuitive than the original one as long as we study "good-enough" quality. There are always more faults to find, and some modules are more critical from an architectural point of view, and these modules contain faults during their whole life cycle.

Using lines of code as a predictor for faults was no success in the original study. Only for pre-release faults, they found weak support for a relation between number of faults and lines of code. In the replication study, we find somewhat stronger support, but it is not consistent across the studied projects. For one of the application projects (Project 3), there was a measure of association between faults and total LOC of around 0.6. For the other application project (Project 1) the measure was around 0.4, while for the platform project, there was no correlation. The correlation between size and fault density is not supported, neither by the original nor by the replication study. Also, the varying results indicate that the hypotheses do not capture the essence of these questions. Factors not taken into account in the studies probably have an impact. The validity of the analysis as such is questioned by Rosenberg [26]; hence it is not surprising that there are no empirical results.

Fault densities are hypothesized to be constant between releases and similar across environments. Both hypotheses are confirmed in the original study, and to some extent also in the replicated study. The results depend on what is meant by "similar". Among the three studied projects, we identify larger dissimilarities between the application projects and the platform project, while the two application projects are closer. Studying the percentage distribution of faults found in Table 3 shows another view of "similarity". Here, all the three projects are "similar". We interpret the data as follows: the development process used in all three projects very much governs the percentage distribution, whereas the technical characteristics of the platform vs. application projects cause different behaviour in terms of fault densities. Hence we confirm the hypotheses, although disclaiming regarding the definition of "similarity".

In summary, the replicated study differs a lot from the original study with respect to size, type of application, number of faults etc., and there is a substantial variation between the three studied projects. Still we add to confirmation of quite general statements. Hence this indicates that there are some general underlying phenomena that can be observed and possibly formulated as theories, although on a very general level [29], which do not seem sufficient for predictive purposes.

6. Conclusions

The motivation for conducting this study was to continue to build empirical knowledge of software fault behaviour in large complex software systems. Others have accomplished the same investigations, and the same hypotheses have been investigated before. Nevertheless, doing research such as a replication is a major cornerstone of science [24]. Given the relatively primitive stage of our understanding of software development processes and the associated fault distributions, attempts to either confirm or reject the structures and behaviour that are published in unique studies, have been our strategy. Without replications, no effort could be put in trying to generalize the findings. Unfortunately, published studies in the investigated area are rather rare. Probably, partially because of the effort necessary to do such an investigation is substantial, but also because replications are not really acknowledged as important research and thereby an attractive proposition. A repeated evaluation could provide

meaningful results for a real-world setting, but still the software engineering field has difficulties in recognizing the benefits [14].

The generalizations of results have to be accomplished step by step. Our goal is not do a complete generalization of the findings, even though for several of the investigated hypotheses, similar results are obtained from our study as the original study. The purpose of the replication has been to investigate variations of the original study. Thus, the same hypotheses (with one exception, which is left out) are investigated, but in a different context and development environment. The generalization of the findings can be accomplished by gradually expanding the context. The investigated projects in the replication study belong to the same sector, the telecommunication domain, as the system from the original study, although the type of system is not the same. In the original study a switching system was developed, and in the replication study the examined projects developed consumer products for the open market. Also, the sizes of the examined systems differ, as well as the programming languages used for development. That this, for the replication study we have kept some parameters unchanged, and varied others. The variations between the studies are considered important to further increase the understanding of fault distributions.

It is shown that the results obtained in some of the hypotheses are not the same in the original study as the replication study. Not even for the three projects included in the replication study the results are identical. Clearly, the dissimilarities indicate that the variations in the research study, regarding the software system under investigation and its development environment, influence the results to a large extent. In those cases the hypotheses do not sufficiently reflect the affecting factors. Nevertheless, we believe the results are of interest to a wider community. By this, we want to contribute to the empirical and scientific basis of this kind of data. By publishing the empirical data that either supports or rejects both conventional wisdom and the results from the original study, more lessons could be learned on software fault distributions.

7. References

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

[2] Andersson, C. and Runeson, P., "A Spiral Process Model for Case Studies on Software Quality Monitoring – Methods and Metrics", to appear in *Software Process: Improvement and Practice*, 2006.
[3] Basili, V. R. and Perricone, B. T., "Software Errors and Complexity: An Empirical Investigation", *Comm. ACM*, 27(1): 42-52, 1984.
[4] Compton T.B. and Withrow C., "Prediction and Control of ADA Software Defects", *Journal of Systems and Software*, 12:199-207, 1990.
[5] El Emam, K., Benlarbi, S., Goel, N., Mela, W., Lounis, H. and Rai, S. N., "The Optimal class Size for Object-Oriented Software", *IEEE Transactions on Software Engineering*, 28(5): 494-509, 2002.
[6] Fenton, N. E. and Ohlsson, N., "Quantitative Analysis of Faults and Failures in a Complex Software System", *IEEE Transactions on Software Engineering*, 26(8): 797-814, 2000.
[7] Fenton, N. E. and Pfleeger, S. L., *Software Metrics: A Rigorous and Practical Approach*, Thomson Computer Press, 1996.
[8] Hatton, L., "Reexamining the Fault Density -Component Size Connection.", *IEEE Software*, 14(2): 89-97, 1997.
[9] Institute of Electrical and Electronics Engineers, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990, 1990.
[10] Juran, J. M. and Gryna Jr. F. M., *Quality Control Handbook* (4th ed.), McGraw Hill, 1988.
[11] Kaaniche, M. and Kanoun, K., "Reliability of a Commercial Telecommunications System", *Proceedings of 7th International Symposium of Software Reliability Engineering*, pp. 207-212, 1996.
[12] Lindsay, R. M. and Ehrenberg, A. S. C., "The Design of Replicated Studies", *The American Statistician*, 47(3): 217-228, 1993.
[13] Malaiya, Y. K. and Denton, J., "Module Size Distribution and Defect Density,", *Proceedings of 11th International Symposium of Software Reliability Engineering*, pp. 62-71, 2000.
[14] Miller, J., "Replicating software engineering experiments: a poisoned chalice or the Holy Grail", *Information and Software Technology*, 47(4): 233-244, 2005.
[15] Munson, J. C. and Khoshgoftaar, T. M., "The Detection of Fault-Prone Programs", *IEEE Transactions on Software Engineering*, 18(5):423-433, 1992.
[16] Möller, K-H. and Paulish, D. J., "An Empirical Investigation of Software Fault Distribution", *Proceedings of 1st International Software Metrics Symposium*, pp. 82-90, 1993.
[17] Ohlsson, N. and Alberg, H., "Predicting Fault-Prone Software Modules in Telephone Switches", *IEEE Transactions on Software Engineering*, 22(12): 886-894, 1996.
[18] Ostrand, T. J. and Weyuker, E. J., "The Distribution of Faults in a Large Industrial Software System", P*roceedings of the 2002 ACM Sigsoft International Symposium on Software Testing and Analysis,* 27(4): 55-65, 2002.
[19] Ostrand, T. J., Weyuker, E. J. and Bell, R. M., "Predicting the Location and Number of Faults in Large Software Systems", *IEEE Transactions on Software Engineering*, 31(4): 340-355, 2005.

[20] Pfleeger, S. L. and Hatton, L., "Investigating the Influence of Formal Methods", *IEEE Computer*, 30(2):33-43, 1997.
[21] Pfleeger, S. L., "Soup or Art? The Role of Evidential Force in Empirical Software Engineering", *IEEE Software*, 22(1): 66-73, 2005.
[22] Pickard, L. M., Kitchenham, B. A, and Jones, P. W., "Combining empirical results in software engineering", *Information and Software Technology*, 40(14): 811-821, 1998.
[23] Reh, J. F., "Pareto's Principle - The 80-20 Rule", *Business Credit*, 107(7): 76, 2005.
[24] Robson, C., *Real World Research* (2nd ed.), Blackwell Publisher, 2002.
[25] Roper, M., Wood, M. and Miller, J., "An Empirical Evaluation of Defect Detection Techniques", *Information and Software Technology*, 39(11): 763-775, 1997.
[26] Rosenberg, J., "Some Misconceptions About Lines of Code", *Proceedings of 4th International Software Metrics Symposium*, pp. 137-142, 1997.
[27] Runeson, P., Andrews, A., Andersson, C., Berling, T. and Thelin, T., "What Do We Know About Defect Detection Methods?", accepted for publication in *IEEE Software*, 2006.
[28] Withrow, C., "Error Density and Size in Ada Software", *IEEE Software*, 7(1): 26-30, 1990.
[29] Zendler, A., "A Preliminary Software Engineering Theory as Investigated by Published Experiments", *Empirical Software Engineering*, 6(2): 161-180, 2001.

PAPER VI

A Replicated Empirical Study of a Selection Method for Software Reliability Growth Models

Carina Andersson

Submitted, 2006.

VI

Abstract

Replications are commonly considered to be important contributors to investigate the generality of empirical studies. By replicating an original study it may be shown that the results are either valid or invalid in another context, outside the specific environment the original study was launched in. The results of the replicated study show how much confidence we possibly could place in the original study.

We present a replication of a method for selecting software reliability growth model to make stop test decisions. We have applied the suggested procedure for the selection method in an empirical study, conducted in a different development environment than the original study. The results of the replication study show that with the changed values of stability and curve fit, the selection method works well on the empirical system test data available, i.e. the method was applicable in another environment than the original one. Also, the application of the SRGMs on function test failures resulted in predictions with low relative error, thus providing a useful approach in giving good estimates of the total number of failures to expect in function test.

1. Introduction

To estimate the reliability of a software system as it undergoes changes through the removal of defects, the theory of reliability growth models have been applied and many software reliability growth models (SRGMs) have been proposed. The reliability of software, as one of the most important attributes of software quality, is closely connected with defects, and is assumed to grow due to the detected defects are corrected and removed from the software. To estimate the remaining number of defects in a software system, which is under test, SRGMs can be applied and guide test management in their decisions in whether continue or stop testing. This paper reports on a replication of a study, originally conducted by Stringfellow and Andrews [20], where a method for selecting SRGMs were suggested and applied to make these release decisions.

The importance of replicating research studies has the last years received a growing awareness in the empirical software engineering community. A finding could not be established as the "truth", based on a single study, since small variations in the execution of a study can have large effect on the results. Nevertheless, to understand the fundamental principles behind the software development phenomena studied, an attempt to run an exact replication is most often not feasible [17], and not even ideal. Instead, replications in a variety of environments are a basis for obtaining more robust and generalizable results. Miller [12] discusses the topic and argues that to receive meaningful results, a step towards families of studies investigating a single hypothesis is necessary. Miller has a focus on replications of controlled experiments, although replications can also be applicable in other forms of studies, to either produce support to a particular theory, or question the claims from the original study.

Hence, the goal of this study is to explore the applicability of the results of an original study in a different context, rather than repeating the study under the same conditions to verify the exact results. Stringfellow and Andrews [20] proposed a selection method for determining the most appropriate software reliability growth model in terms of predictive ability, stability and curve fit. SRGMs are in the study chosen to estimate the remaining number of defects in a software system of interest, to help management in release-decisions during testing. In a case study the method is empirically evaluated on data from three releases of a large medical record system. The results of the original study show that the

selection method worked well on the three data sets presented in the case study, although several underlying assumptions of the SRGMs were violated, when applying the models on the real-world data.

A replication of the study by Stringfellow and Andrews is presented in this paper. The proposed selection method is implemented and the presented approach is applied in a different environment to evaluate its usefulness and applicability. That is, in a new case study the method's applicability is validated, with failure data from three telecommunication software system projects. The replication is one step towards a generalization of the selection method. However, it is important to notice that one replication is not enough. In the study, some parameters are changed, compared to the original study, while the basic ideas of the selection method are maintained. The selection method is in this case applied to a number of failures which is approximately 10-fold larger, while the projects' lead-times are once or twice of the length compared to the original study. This brought us to the conclusion that the evaluation criteria in terms of the given values of stability and curve fit used in the original study could not be transferred to this new context without adaptation. In addition, the study by Stringfellow and Andrews examined the method with failure data from system test, while in this study the selection method is applied to failure data from system test, but also to failure data from function test to investigate whether useful predictions may be made already during this test activity.

The paper is structured as follows: Section 2 describes SRGMs and their underlying assumptions. Also, the original study by Stringfellow and Andrews is discussed. Section 3 describes our approach of the replication study, presents the failure data and the differences compared to the original study. In addition, the findings compared to the results of the original study are presented. Finally, in Section 4 the conclusions are presented.

2. Background

Quality, cost and schedule have been declared as the most important software project characteristics [14]. The latter two are quantitatively measurable, while quality is more difficult. Software quality has a wide range of attributes, such as functionality, usability, portability, and maintainability [8], which results in an absence of one concrete measure

for software quality. Software reliability, however, could be seen as a key factor in software quality, since it quantifies software failures. Software reliability is defined as *the probability of failure-free operation* of a software program for a *specified time* in a *specified environment* [14].

Several different definitions of error, fault and failure exist in the literature. A crucial part in applying software reliability measurements concerns separating these definitions. In this paper we use the following terminology, as defined by IEEE [7]: an *error* of commission or omission causes a *fault* in the code, which in turn manifests itself as a *failure* that can be observed during software testing or operation. The program has to be executed for the failure to occur and to reveal the departure of the results from the program operation from the requirements. Hence, the failure is something dynamic. The failure occurrence process is of most interest when considering reliability quantities. However, the failure behaviour is obviously affected by the number of faults existing in the software being executed [14].

According to the definition of reliability given above, another aspect of software reliability measurement is time. The reliability quantities are related either to the execution time for a software system, which is the CPU time actually spent by the computer executing the software, or the calendar time. The third aspect of the software reliability definition concerns the execution environment. The environment is described by the operational profile. Musa et al. [14] describes the concept of operational profiles. An operational profile consists of the set of operations that a system is designed to perform and their probabilities of occurrence. Thereby, a quantitative characterization of how the system will be used is provided.

2.1 Software reliability growth models

The assumptions of SRGMs are generally stating that the models are applicable during the system test, where cycles of test executions, observed failures, repair, and continued testing are repeated. By observing the changes in failure rate over time, management has a support in using these to make decision about when to stop testing. Practical experiences of the use of reliability growth models in a variety of contexts are published, e.g. by Musa and Ackerman [15], Ehrlich et al. [2], Wood [21][22], and Jeske and Zhang [9].

Several SRGMs assume that the observed failures occur as a random process, a non-homogenous Poisson process (NHPP), which means the failure intensity is not constant. As faults are detected and removed from the software, it is expected that the observed number of failures per time unit will decrease. The expected number of failures observed by time t is given by $\mu(t)$, with the bounded condition $\mu(\infty) = a$, where a is the expected number of failures to be observed eventually. Four common SRGMs are used in the original study, the basic Musa or Goel-Okumoto (G-O) model [6][14], the delayed S-shaped model [23], the Gompertz model [10], and the Yamada exponential model [24]. These models are all based on a NHPP. The same models are used in this replication study. Table 1 gives an overview of the models.

Table 1. SRGMs used in this study.

Model	Type	Equation $\mu(t)$	Reference
G-O	Concave	$a(1-e^{-bt}),\ a \geq 0,\ b > 0$	Goel and Okumoto [6]
Delayed S-shaped	S-Shaped	$a(1-(1+bt)e^{-bt}),\ a \geq 0,\ b > 0$	Yamada et al [23]
Gompertz	S-Shaped	$a(b^{c^t}),\ a \geq 0,\ 0 \leq b \leq 1,\ c > 0$	Kececioglu [10]
Yamada	Concave	$a(1-e^{-bc(1-e^{-dt})}),\ a \geq 0,\ bc > 0,\ d > 0$	Yamada et al [24]

SRGMs can be classified into two major classes, concave and s-shaped models [21]. The concave models assume a pattern of decreasing failure rate, while the s-shaped models assume that early testing is not that efficient as later testing. An s-shaped growth curve may reflect the initial learning curve at the beginning of the test process, as test teams become familiar with the software system and its testing procedures, followed by growth where the failure detection rate increases, and finally leveling off when the remaining faults become fewer and more difficult to detect.

Modeling the software failure process can hardly be expected to be precise, and underlying assumptions are necessary for each model, some more reasonable than others. The assumptions for each model should be evaluated in terms of the test environment the failure data is obtained from. Wood [22] gives an overview of a list of assumptions and discusses their accuracy for the test environment used in his study. For example, in general all software reliability models assume that the defect detection occurs during testing that follows an operational profile [11][16]. In addition, with some exceptions, for example presented by Fujiwara and

Yamada [4], most reliability models assume a perfect debugging environment, i.e. the defects are assumed to be corrected immediately and without inserting any new faults. In practice, obviously software faults may not always be fixed during debugging, and when the software system is changed, new faults may be introduced.

Although several of the SRGMs' assumptions might be violated, the models could be applied with a corresponding curve to a set of data points representing the cumulative number of failures. The fit to the data set could be examined by evaluating the deviation between the observed cumulative number of failures and the fitted values, by using a statistical test, e.g. χ^2, Kolmogorov-Smirnov or R^2. Gaudoin et al. [5] evaluated the power of several of this type of tests, applied to a number of reliability models, showing that for example the simple R^2-test had as much power as the other tests and in some occasions more. More information on the specific tests can be found in literature by Siegel and Castellan [18] and Montgomery [13].

2.2 The original study

The original study, conducted by Stringfellow and Andrews [20], describes an approach applying several SRGMs to cumulative failure data grouped by week to select the model(s) that best fit the data. By a good fit of a model, more accurate predictions of total number of failures are expected, thereby giving a decision support for whether to stop testing and release the software product or to continue testing for another week. The predictions are based on test time and failure data. Stringfellow and Andrews used calendar time for the reliability quantity test time, since other possibilities were not available for the examined failure data sets. The failure data consisted of defect reports, reported during system test.

The four SRGMs used in the study, the basic Musa or Goel-Okumoto (G-O) model [6][14], the delayed S-shaped model [23], the Gompertz model [10], and the Yamada exponential model [24], were selected because they represent a range of assumptions. Important to consider when choosing which models to apply, is simplicity. To incline practitioners to use a SRGM, it has to be simple in concept and inexpensive to collect the input data required. A user without extensive mathematical background should be able to understand and apply the model [16], which relates rather well to the models in this study.

Three model evaluation criteria are used in the original study, a *goodness of fit measure* (GOF), the *prediction stability*, and the *predictive ability*. Stringfellow and Andrews have chosen to base their GOF measure on the simple R^2 test. The choice is motivated by Gaudoin et al. [5], who evaluated the power of several statistical tests for GOF for a variety of reliability models. The larger the R^2 value, the better the fitted equation explains the variation in the data. The evaluation showed that this measure was as least as powerful as the other GOF tests compared. Stringfellow and Andrews chose a threshold of R = 0.95.

In addition to the GOF measure, the models are evaluated in terms of prediction stability. A threshold for the stability is set; the prediction in week *i* should be within 10% of the prediction of week *i - 1*. The threshold value of 10% is subjectively chosen, motivated by a rule of thumb given by Wood [21].

Prediction accuracy is the last model evaluation criterion used in the study. The predictive ability is measured in terms of *error* (estimate - actual) and *relative error* (error/actual).

The proposed method for selecting SRGMs based on the models that best fits the data consists of several steps. Stringfellow and Andrews are giving a detailed description of each step of their approach, while only a short summary is presented below and illustrated in Figure 1.

The procedure for the selection method is executed once a week (on the assumption that the cumulative number of failures are grouped by week), starting with recording the cumulative number of failures found. The next step is considering whether it is appropriate to apply the models. In case only a minor part of the execution of the test plan is complete, it may not be valuable to apply the SRGMs. Stringfellow and Andrews recommend that at least 60% of the planned testing is complete before applying the models, which also is motivated by Ehrlich et al. [1]. Once testing has proceeded this far, the chosen SRGMs are applied to the data at the end of each week, using a commercial curve fit program. The curve fit program attempts to fit each model to the data. A model diverges if no fit can be performed to the data set, and then the model is considered inappropriate and excluded from the selection procedure in future weeks. However, if a fit is obtained and the model converges, the program outputs the model's parameters such as the estimate for the expected number of total failures. GOF of the fitted curve is also evaluated, based on the R^2 value. Models are considered inappropriate for the data set if they have less good fit, i.e. an R value below the threshold value R = 0.95.

These models are excluded from the selection procedure in subsequent weeks. Another criterion could exclude the models from the selection procedure; if the estimate of expected total number of failures given by the curve fit program is lower than the actual number already detected, this could lead to a false sense of security, and will thereby lead to exclusion of the model in subsequent weeks. Thereafter, the model stability is examined with the stability threshold of 10% of previous week's prediction. If no model has stabilized by giving a prediction within the 10% interval, testing should continue and failure data be collected for another week. With at least one stable model, the final step in the approach gives the estimate that is the maximum estimate of all stable models. A decision based on the difference between the estimate and actual number of failures detected will determine whether testing should continue or if the developed software system has reached an acceptable level.

The results from the study by Stringfellow and Andrews show that the selection method worked well. With the chosen threshold values for GOF and stability, at least one model was acceptable by the time testing was approaching a decision point for stopping. The selection method seemed to differentiate between the included models, and the predictions corresponded well with the actual numbers of failures.

3. The replication study

The goal of our study is to replicate the study by Stringfellow and Andrews, and to evaluate their suggested approach for selecting SRGMs to software failure data. We have monitored the trends of failures detected and attempted to predict the testing process with assistance of the selection method of SRGMs. However, as well as Stringfellow and Andrews had to violate some of the underlying assumptions, which the SRGMs are dependent on, so had we when applying the models in practice.

In the following section, our approach of the selection method is presented. A description of the cases study is given, with detailed information of the data we have used. Also, differences from the approach taken by Stringfellow and Andrews are presented.

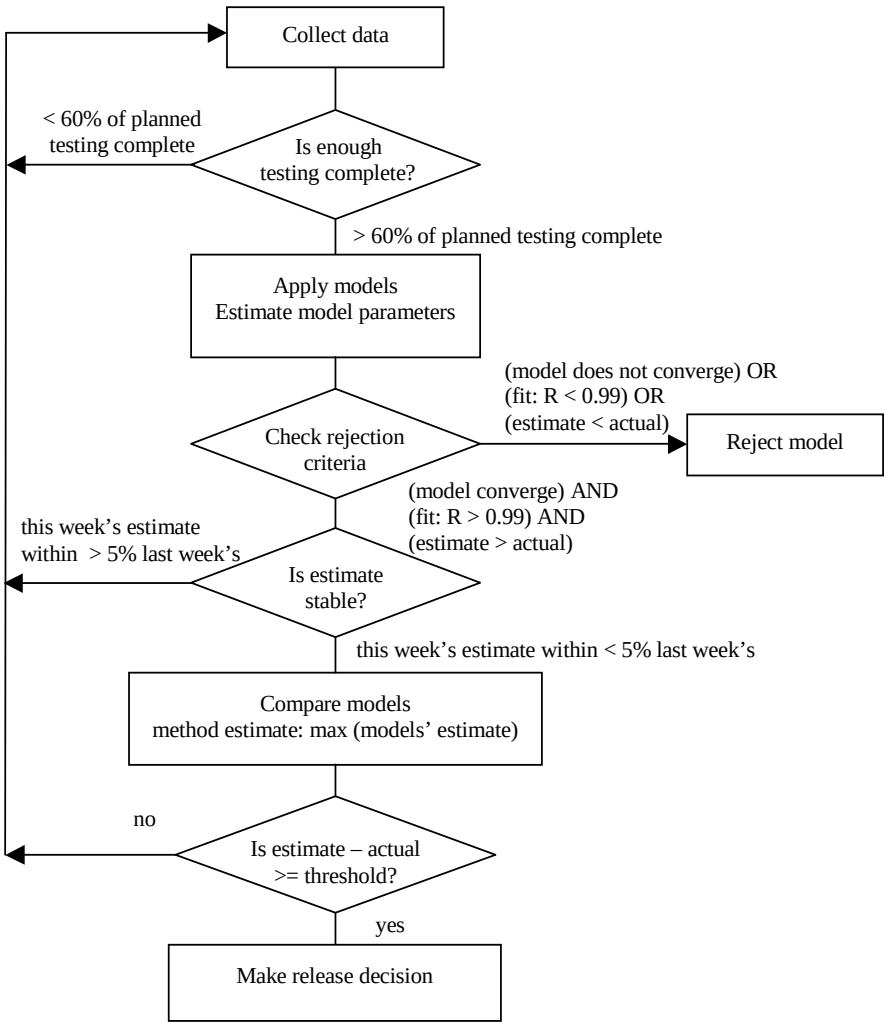

Figure 1. *Flowchart for the selection method. Derived from [20]. The rejection criteria are based on the threshold values from the replication study.*

3.1 Case study

The failure data used in this replication study comes from three software development projects conducted in the telecommunications industry. The development organization has a high market sensitivity, which implies a strong need for early indicators of project progress and specifically fault

content in the developed products, to enable early actions to reduce costs and to plan the test activities more preventively.

The software projects follow an incremental development process, with an extensive number of iterations, with the project duration ranging in a number of months. The development projects are divided into *feature groups*, each having responsibility of a specific part of the functionality to be developed. In each iteration, the feature groups are delivering components of the assigned functionality to the main software system. In addition to the developers in the feature groups, these groups also consist of function testers. The developers do the unit testing themselves, thereafter the function testers take over the testing responsibility of the implemented software.

System test is a test organization separated from the development and function testing feature groups. The system test organization runs its test suites on the latest available version of the product. The testing is performed both in the development environment and in the real operational environment.

Due to the incremental development environment, several activities occur at the same time. An overview of the workflow is given in Figure 2, which illustrates the overlapping activities. Some important milestones used in the development process are marked in the figure. Before *Alpha* the main activities are development and function test, while the main part of system test starts at *Alpha*. At *ship date* the first version of the system is released. After this milestone testing and debugging continues. The versions with the corrected implementations are released at the *update releases (UR)*.

3.2 Data

The data used in this replication study are obtained from a large failure report database. Data from the three software projects is extracted from the database, classified by detection date, and whether the reporter who detected the failure is function tester or system tester. Thereby, the failure reports are separated into function test failures and system test failures, and the data sets are not limited to failures detected by system test as is the case in several other studies [21][20]. We chose to use the full data set available, and the subsets of it obtained through data partitioning into function and system test failures, since this approach enables us to do more detailed analyses. The function test failures are detected in testing

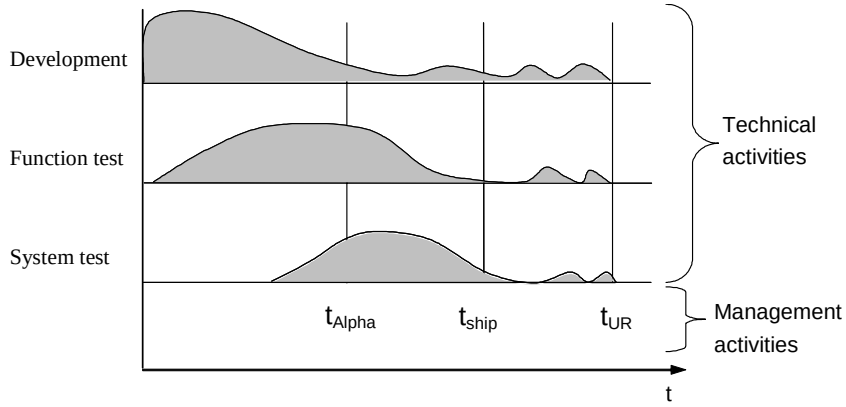

Figure 2. *Project work flow.*

procedures, which are considered to be feature related. Combining this type of data with ordinary reliability growth testing data is quite contrary to what is advisable [11], at least if the test cases are not selected randomly according to an operational profile. In this case, no operational profile is used for test case selection, but on the other hand, the function tests do not proceed sequentially. The feature groups execute function test cases in parallel, each group focusing on its own functionality, but the groups are executing the tests at the same time, resulting in a mixed test suite execution of different features.

The failure data is accumulated per week, using calendar time, since execution time data is not available in the organization. In the development projects, failures were reported in relevance to calendar time, with granularity of days. Hence, we use this scale, which also was motivated by a rather constant test effort per week. However, there were holidays, such as summer vacations and Christmas, when the test effort was less than normal, and also regression test periods when the test effort was more intensive than normal, which constituted a significant percentage of the calendar time. For this reason, a modified time scale is used, which takes into account only testing days and compensates for the non-constant test effort.

Duplicate detection of failures is not included in the data sets, that is, only one failure report is kept per observed failure. If the same failure is detected more than once, only the first is entered into the database. Also, if an underlying fault is causing different failures, only one failure report is kept for the analysis. Hence, only unique faults are represented in the

study. Neither are change requests and problems that not are code faults included in the analysis.

The predictions from the SRGMs, based on the available data sets, are compared to the cumulative number of failures detected after t_{ship}. Hence, failures detected post-ship are also separated into failures detected in function test and system test, i.e. the number of failures detected by each test activity *after* t_{ship}. Thus, to examine the predictive ability, the predictions obtained from the data sets, consisting of failures detected by function test is compared to the actual number of failures detected by the same test activity after t_{ship}. Vice versa goes for system test failures.

3.3 Comparison to the original study

Some differences exist between our study and the one conducted by Stringfellow and Andrews. These differences are presented below, in conjunction with a summary of aspects considered during the application of the suggested selection method.

- Can calendar time be used when the original models assume execution time?

Stringfellow and Andrews investigate the use of calendar time. We also use a sort of calendar time. However, the time used is modified to avoid the reflection of differences in test effort due to holidays and more intensive periods of testing.

- How robust are the models when the underlying assumptions are not met?

As mentioned, when applying SRGMs in practice, several of their stated assumptions get violated, which is the case also with the data in this case study. In this study we can assume an imperfect debugging environment where new faults may be introduced by the correction of detected faults. Neither is the code base held constant, but growing during the test process when the developers iteratively deliver new functionality to the main software system. In addition, we apply the SRGMs on both failure data from system test and function test.

- What is a good fit of the models?

Analogous to Stringfellow and Andrews, we use the R^2 value as GOF measure. However, the number of data points of detected failures is not in

the same range as in their study (see Table 2), which implies that their chosen value is not appropriate in our study. Stringfellow and Andrews base their threshold value on a discussion by Gaudoin et al. [5] who evaluate a different model, although they find the value associated with a high confidence level. Also based on Gaudoin et al. we chose a threshold value R = 0.99, to better reflect the models' appropriateness in this application, since the critical value depends on the number of data points. An alternative to setting a threshold value is to base the selection method on choosing the SRGM that has the highest R^2 value, and exclude the rest. However, the purpose is to maintain the iterative procedure in the selection method. We do want to keep more than one candidate in the selection process, especially those which may give reasonable predictions later. The chosen threshold value is noticeably higher than the value used by Stringfellow and Andrews, R = 0.95, which might not have resulted in any selection at all applied to our data sets. The effect of choosing the threshold value R = 0.99 is further discussed in the subsections presenting the application of the selection method to each data set.

- What is considered a stable model?

Related to the discussion of the GOF measure, with our larger data sets in terms of data points and number of failures, the higher number of detected failures compared to the study by Stringfellow and Andrews implies that the stability measure may not be relevant at the 10% level. We have chosen the threshold at 5% of the prediction of the previous week, to keep the number of models remaining in the selection process on an appropriate level.

- Which estimate is appropriate to use when more than one model is stable?

Stringfellow and Andrews chose the conservative choice, the maximum estimate when more than one model was remaining and giving predictions of existing failures. Often there is a large difference in the estimates given by the different models and we did not find this approach applicable in every case. We have chosen to evaluate the selection method on the same aspect, the highest estimate, as in the original study. However, this should be carefully considered when applying the selection method in an industrial context. We noticed that further reflection on the received estimates from the non-rejected models must be done to avoid unnecessary time spent on testing.

- What should the predicted values be compared to?

Stringfellow and Andrews use the number of failures reported after release as the comparative value to calculate predictive ability and relative error with. Our corresponding value for the calculation of relative error is based on the number of failures detected until t_{UR} according to Figure 2, separated into the two categories of failures detected by system test and function test. Since new corrected code is implemented after t_{ship}, function test failures continue to occur after this milestone. The parallel activities of system test and function test cause us to take this approach, although the main portion of function test is conducted before system test, and function test failures are mainly assumed to not be detected by system test. However, the size of the fraction of failures that could be detected both by system test and function test is not known, but in this study is believed to be small.

Table 2. Project data for the original study and the replication study. FT = function test, ST = system test.

	Original study			**Replication study**		
	Release 1	**Release 2**	**Release 3**	**Project 1**	**Project 2**	**Project 3**
Duration ST (weeks)	18	17	13	22	27	25
Duration FT[a] (weeks)	-	-	-	35	46	41
Number of failures (ST)	231	245	83	585	1330	3839
Number of failures (FT)	-	-	-	2704	4802	5343

a. The function test runs in parallel with the system test in the replication study. I.e. the end dates are identical, although the system test starts after the function test and the duration in number of weeks are shorter.

3.4 Results

In this section we present the results from the replication study. In Section 3.4.1 the selection method for SRGMs is replicated on system test failures, while Section 3.4.2 presents the new approach where the SRGMs are applied to the data sets of function test failures.

3.4.1 System test failures

In the following we present the data from the three projects while applying the G-O, delayed S-shaped, Gompertz, and Yamada models to failure data detected by system test, starting at 60% of the planned testing and ending at the ship date. The columns show the test week, the cumulative number of failures found, and for each model: the prediction of total number of failures and the adherent GOF-value (R-value). An *S* indicates that the model is stabilizing in the specific week, while a *D* indicates a destabilization of the model. An *R* indicates the model is rejected in the selection method and not considered as an appropriate model in future weeks.

The tables show the prediction data when applied to the failure data detected in system test. Notice that test week starts counting from the beginning of system test; in parentheses the function test week is given.

Project 1 system test data and the application of the SRGMs are shown in Table 3. Both concave models are rejected because they do not converge in test week 16. The rejection is expected when inspecting the cumulative failure curve, which clearly is s-shaped. Both models do converge later on, although they give very high estimates and have R-values well below the threshold value of 0.99.

The delayed S-shaped model and the Gompertz model have better fits to the curve, although the delayed S-shaped model do not stabilize until week 21, while the Gompertz model stabilizes in week 17, destabilizes the week after and stabilizes once more in week 19. Figure 3 illustrates the cumulative failure curve and the s-shaped models' estimates of total number of failures for each week, starting week 16. The appearance of the failure curve from system test in project 1 is obviously rather difficult to fit a model to. According to the original selection method, in test week 22 the delayed S-shaped model is favored, since it gives a higher estimate than the Gompertz model, although its R-value is lower than the Gompertz model's.

Compared to the total number of failures reported, the predicted value of the delayed S-shaped model had a relative error of -0.031, while the Gompertz model had a relative error of -0.357, despite the high value for the curve fit, see Table 4.

Table 3. Predicted total number of failures for project 1 (ST).

Test week ST (FT)	Failures found	G-O		Delayed S-shaped		Gompertz		Yamada	
		Estimate	R-value	Estimate	R-value	Estimate	R-value	Estimate	R-value
16 (29	226	-	- (R)	1020	0.9963	421	0.9976	-	- (R)
17 (30)	249	-	-	948	0.9970	423 (S)	0.9981	-	-
18 (31)	252	82 300	0.9637	758	0.9965	383 (D)	0.9980	-	-
19 (32)	265	151 000	0.9680	661	0.9963	367 (S)	0.9982	-	-
20 (33)	279	89 300	0.9717	609	0.9965	364	0.9984	-	-
21 (34)	297	90 500	0.9747	590 (S)	0.9969	373	0.9986	127 200	0.9748
22 (35)	306	114 000	0.9775	567	0.9971	376	0.9987	49 830	0.9774

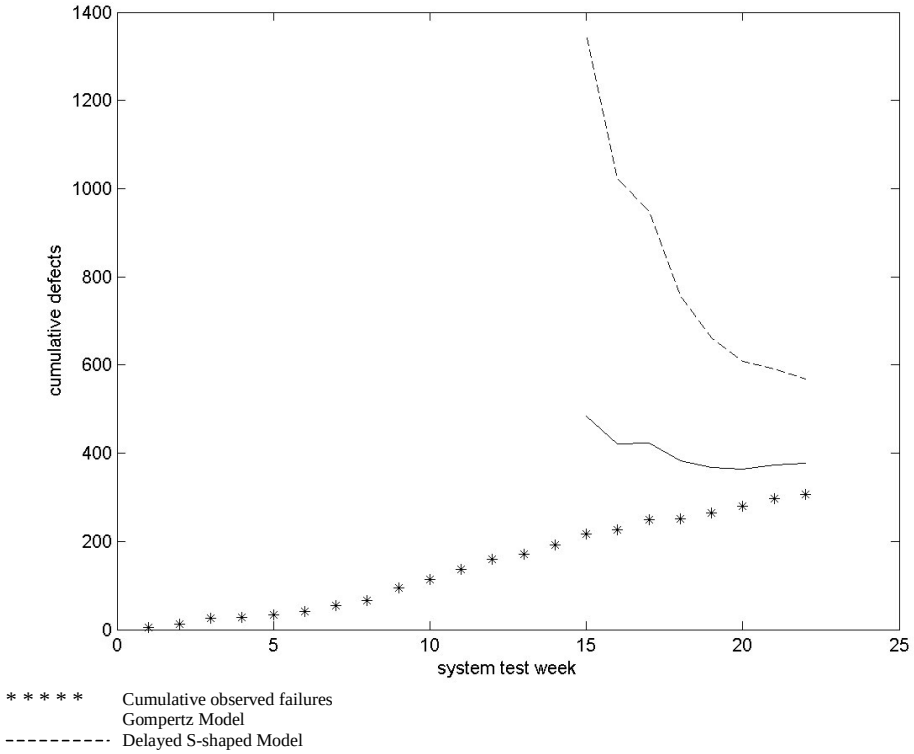

Figure 3. *Plot of project 1 data (ST) and each week's prediction of total number of failures, from SRGMs not rejected.*

Table 4. Final estimates and error by SRGMs not rejected for project 1 (ST) at week 22.

Model	Estimate (true value: 585)	R-value	Error	Relative error
Delayed S-shaped	567	0.9971	-18	-0.031
Gompertz	376	0.9987	-209	-0.357

Project 2 system test data and the applied SRGMs are shown in Table 5. When applying the models to the data set, the concave models give very high estimate of the total number of failures in the beginning of the presented test period, but have R-values above the threshold value.

During the subsequent weeks, the models are lowering their estimates to more reasonable figures, and finally also stabilize in week 27.

The delayed S-shaped model and the Gompertz model stabilize in test week 18, with good curve fit and high R-values for both models. The delayed S-shaped model, which gives the highest estimate of the two, predicts the total number of failures to be 1110 in week 18, to compare with the actual number of failures detected, 859. If the difference between the actual number and the prediction is considered too large, the decision to continue testing should be taken. Testing did proceed several weeks more.

In week 27, when the G-O model and the Yamada model stabilize, the original selection method recommends the conservative choice, to follow the maximum estimate. In this case it is the G-O model's prediction, which is 2320 failures to compare with the actual number of detected failures of 1100. The s-shaped models estimated the total number of failures to 1160 and 1180, rather close to the actual number of detected failures, and the estimated values had been stable for several weeks, see Figure 4. In Figure 4 the predictions of the s-shaped models are shown from week 11, and as seen these stabilize already in week 15 (not presented in Table 5, since 60% of the planned testing was not completed at that time). Figure 4 also shows the predictions of the G-O model and the Yamada model (giving nearly the same estimates), starting in week 23.

After week 27 and ship date, the total number of failures detected was 1330 - 1100 = 230. An amount well below the prediction from the G-O model, but also a little higher than the predictions from the s-shaped models. The values of relative error are presented in Table 6, where also the R-values are presented. These indicated good curve fit for each model, although the predictions were not very good.

Table 5. Predicted total number of failures for project 2(ST).

Test week ST (FT)	Failures found	G-O		Delayed S-shaped		Gompertz		Yamada	
		Estimate	R-value	Estimate	R-value	Estimate	R-value	Estimate	R-value
17 (36)	822	156 000	0.9964	1100	0.9970	1080	0.9981	137 900	0.9965
18 (37)	859	167 000	0.9966	1110 (S)	0.9974	1080 (S)	0.9983	52 770	0.9966
19 (38)	880	33 300	0.9961	1110	0.9977	1070	0.9985	10 690	0.9961
20 (39)	899	7280	0.9954	1110	0.9979	1060	0.9986	5774	0.9954
21 (40)	925	4420	0.9949	1110	0.9981	1060	0.9987	4406	0.9950
22 (41)	950	3370	0.9946	1110	0.9983	1070	0.9989	3373	0.9946
23 (42)	992	2980	0.9948	1120	0.9983	1080	0.9988	2979	0.9948
24 (43)	1027	2770	0.9951	1140	0.9982	1110	0.9986	2767	0.9951
25 (44)	1043	2560	0.9952	1150	0.9982	1120	0.9986	2557	0.9952
26 (45)	1069	2410	0.9954	1160	0.9981	1140	0.9985	2409	0.9954
27 (46)	1100	2320 (S)	0.9957	1180	0.9980	1160	0.9983	2318 (S)	0.9957

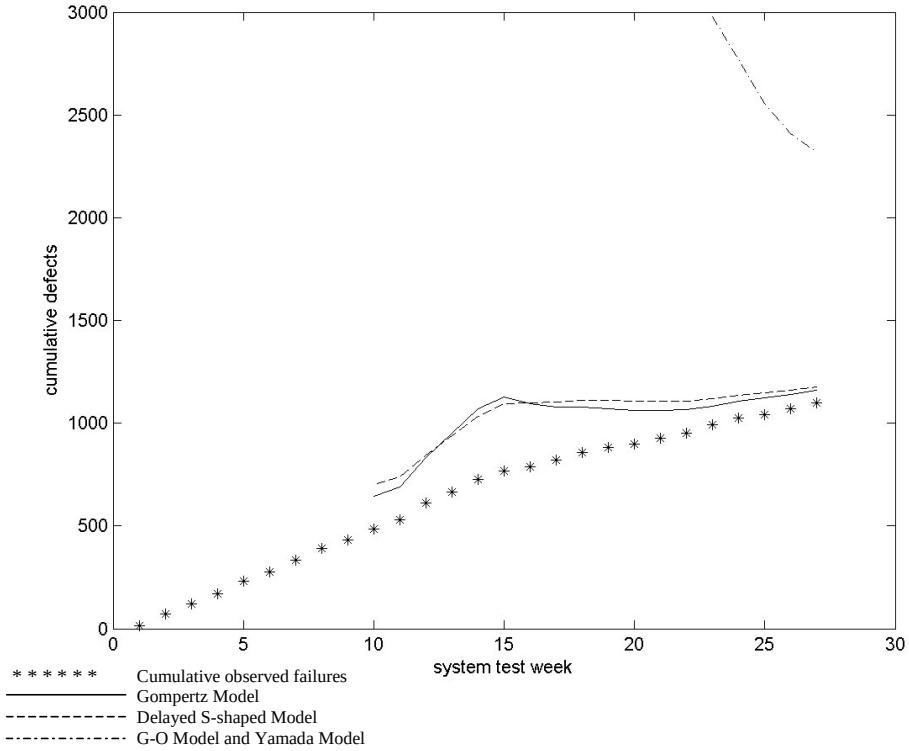

Figure 4. *Plot of project 2 data (ST) and each week's prediction of total number of failures, of SRGMs not rejected.*

Table 6. Final estimates and error by SRGMs not rejected for project 2 (ST).

Model	Estimate (true value: 1330)	R-value	Error	Relative error
G-O	2320	0.9957	990	0.744
Delayed S-shaped	1180	0.9980	-150	-0.113
Gompertz	1160	0.9983	-170	-0.128
Yamada	2318	0.9957	988	0.743

Project 3 system test data and the application of SRGMs are shown in Table 7. The selection method did not reject any of the SRGMs for the data. However, already at week 17 the two concave models have good curve fits, with the highest R-values (the G-O model and the Yamada model have very similar values both for predictions and curve fit for this data set). On the other hand, at the end of the system test period, in week 25 the Gompertz model has the highest R-value, indicating that a selection procedure based on just following the highest R-value would give a different result.

Figure 5 shows the cumulative failure curve of the failures detected by system test in project 3 and the four models' predictions for each week. The estimates from the s-shaped models start in week 18, when they stabilize. The G-O model and the Yamada models are shown from test week 23.

When reaching week 25, when the two concave models have stabilized, the original selection method recommends following these models, as they give the higher estimates (at week 25 the G-O model and the Yamada model are giving the exact same estimate). Making a decision to stop testing based on the estimates from the concave models, would probably have resulted in continued testing. At week 25, only 3263 failures had been found, while the concave models predicted a total of 5060 failures.

All models had at week 25 fairly good R-values. Another 576 failures were reported after test week 25. Thereby, the concave models overestimated. Meanwhile, the s-shaped models underestimated. The values for relative error are presented in Table 8.

For this third system test data set, the failure data from project 3, the original selection method suggested the concave models to be followed. Nevertheless, it was later shown that these models had a larger relative error than the s-shaped models. The Gompertz model's prediction was closest to the actual number of failures, and did also have the best fit according to the high R-value, but was as in previous cases underestimating.

Table 7. Predicted total number of failures for project 3 (ST).

Test week ST (FT)	Failures found	G-O		Delayed S-shaped		Gompertz		Yamada	
		Estimate	R-value	Estimate	R-value	Estimate	R-value	Estimate	R-value
17 (33)	2707	7750	0.9989	3000	0.9916	3300	0.9967	7752	0.9989
18 (34)	2824	7330	0.9989	3090 (S)	0.9920	3380 (S)	0.9971	7325	0.9989
19 (35)	2905	6860	0.9989	3160	0.9924	3420	0.9973	6859	0.9989
20 (36)	2993	6490	0.9989	3230	0.9927	3460	0.9976	6348	0.9989
21 (37)	3050	6110	0.9988	3270	0.9931	3480	0.9978	6106	0.9988
22 (38)	3088	5730	0.9985	3300	0.9936	3480	0.9980	5733	0.9985
23 (39)	3151	5460 (S)	0.9983	3330	0.9939	3490	0.9981	5455 (S)	0.9983
24 (40)	3196	5220	0.9980	3360	0.9945	3490	0.9983	5223	0.9980
25 (41)	3263	5060	0.9980	3380	0.9945	3510	0.9983	5060	0.9979

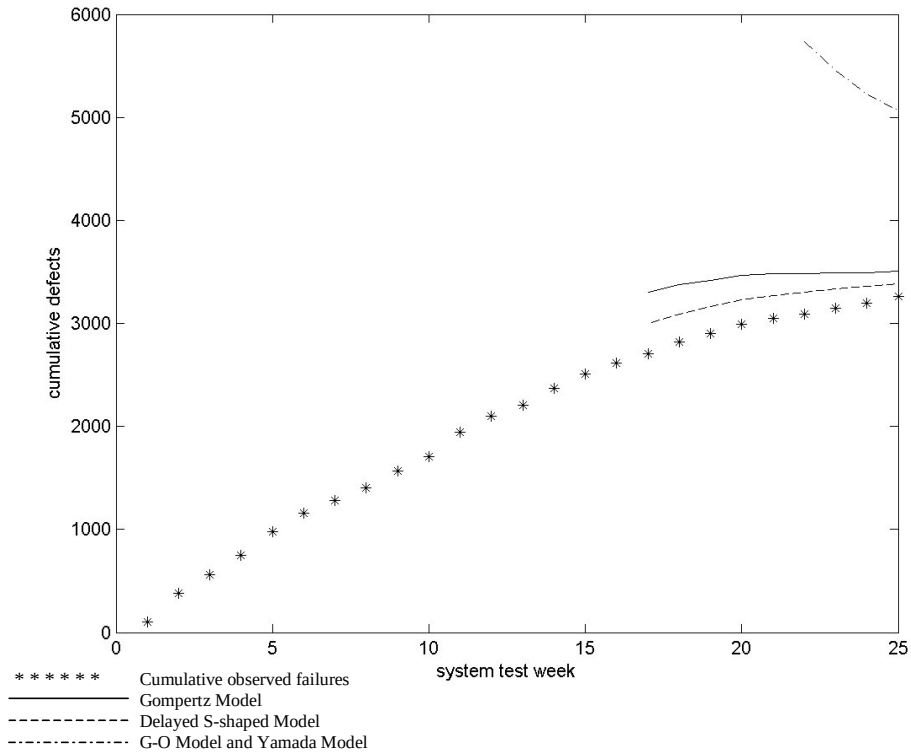

Figure 5. *Plot of project 3 data (ST) and each week's prediction of total number of failures, from SRGMs not rejected.*

Table 8. Final estimates and errors by SRGMs not rejected for project 3 (ST).

Model	Estimate (true value: 3839)	R-value	Error	Relative error
G-O	5060	0.9980	1221	0.318
Delayed S-shaped	3380	0.9945	-459	-0.120
Gompertz	3510	0.9983	-329	-0.086
Yamada	5060	0.9979	1221	0.318

3.4.2 Function test failures

In the following we present the data from the three projects and the application of the G-O, delayed S-shaped, and Gompertz models to failure data detected by function test. The data sets, consisting of the cumulative failures detected in function test, were all following an s-shaped curve, clearly seen by the first visual inspections. These s-shaped data sets could not be fitted to the concave Yamada model, and the model did not converge in any of the cases using function test data. Thus, the model is not included in the following section. The concave G-O model did not perform very well either for the data from function test, though is included to illustrate how the results from a concave model differ from the s-shaped models'.

Project 1's failure data from function test and the estimates from the SRGMs are presented in Table 9. The delayed S-shaped model and the Gompertz model perform well, which are indicated by the R-values. In Figure 6, the plot of the failure data from project 1 is shown, together with the estimates for each week for the delayed S-shaped model and the Gompertz model. Due to the rather obvious s-shaped curve of the

Table 9. Predicted total number of failures for project 2(ST).

Test week FT	Failures found	G-O		Delayed S-shaped		Gompertz	
		Estimate	R-value	Estimate	R-value	Estimate	R-value
22	1460	104 000	0.9429 (R)	6170	0.9914	1750	0.9988
23	1495	107 000	0.9485	4850	0.9910	1740 (S)	0.9989
24	1575	173 600	0.9536	4240	0.9912	1760	0.9990
25	1657	107 000	0.9576	3940	0.9918	1810	0.9989
26	1707	120 000	0.9614	3690	0.9923	1860	0.9988
27	1761	119 000	0.9648	3510 (S)	0.9943	1900	0.9987
28	1800	107 000	0.9677	3350	0.9931	1940	0.9986
29	1826	128 000	0.9703	3200	0.9933	1960	0.9987
30	1861	106 000	0.9723	3070	0.9934	1990	0.9987
31	1898	103 000	0.9741	2980	0.9936	2010	0.9987
32	1915	122 000	0.9756	2880	0.9936	2020	0.9987
33	1940	112 000	0.9765	2810	0.9937	2030	0.9988
34	1971	106 000	0.9772	2750	0.9938	2050	0.9988
35	2012	103 000	0.9779	2700	0.9939	2070	0.9988

cumulative failures, the concave models fit less well. The G-O model was rejected as appropriate model, due to a low R-value, 0.9429, well below the threshold value (both our and the value chosen by Stringfellow and Andrews). The Yamada model, not shown in the table, was rejected since the model did not converge in week 22, indicating that a good fit will not be obtained later on either.

Figure 6 illustrates how the Gompertz model is smoothly following the cumulative failure curve, while the delayed S-shaped is closing in from higher estimates. Choosing the less conservative alternative to trust the lower predicted values instead of the highest, in this case from the Gompertz model, would result in an expected number of failures too low. Considering the failure detection after week 35, the Gompertz model underestimated and had a relative error -0.234, see Table 10, even though the model had the best curve fit. The estimate from the delayed S-shaped model in week 35 was approximately not different from the actual number of failures. Thus, the original selection method would choose the most appropriate model for the prediction of total number of failures to expect in function test.

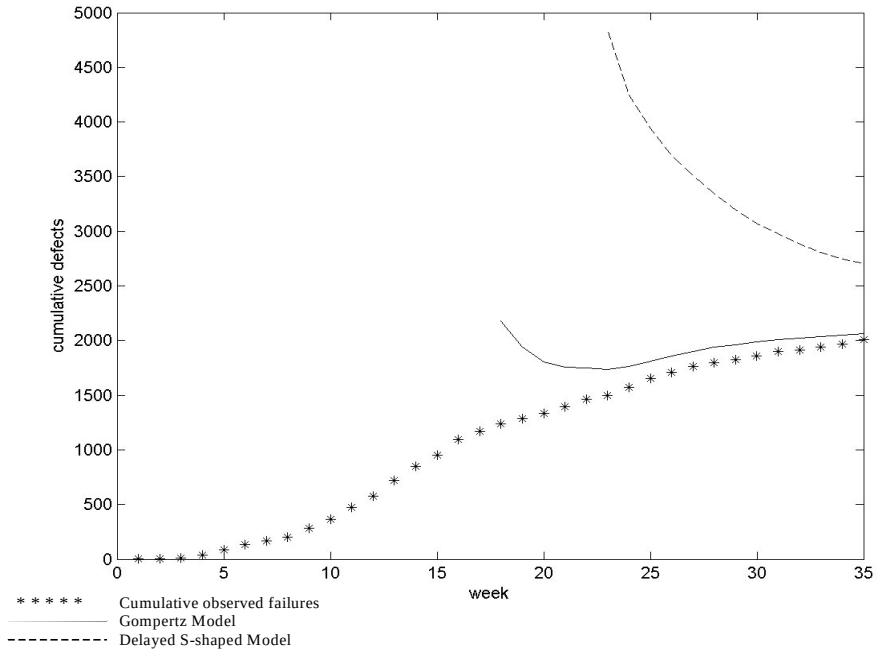

Figure 6. *Plot of project 1 data (FT) and each week's prediction of total number of failures, from SRGMs not rejected.*

Table 10. Final estimates and error by SRGMs not rejected for project 1 (FT).

Model	Estimate (true value: 2704)	R-value	Error	Relative error
Delayed S-shaped	2700	0.9939	-4	-0.002
Gompertz	2070	0.9988	-634	-0.234

Project 2's failure data from function test and the estimates from the SRGMs are presented in Table 11. According to the selection method, again the concave models were rejected as appropriate models. In this case the G-O model was rejected because the model does not reach our threshold value for stability.

The two s-shaped models both stabilize as early as week 31, with high R-values for both models. The delayed S-shaped model could be seen to be very stable, especially the last 9 weeks, with estimates ranging from

Table 11. Predicted total number of failures for project 2(ST).

Test week FT	Failures found	G-O		Delayed S-shaped		Gompertz	
		Estimate	R-value	Estimate	R-value	Estimate	R-value
30	3453	165 000	0.9928	4490	0.9995	3890	0.9987
31	3537	148 000	0.9932	4530 (S)	0.9995	3960 (S)	0.9986
32	3645	151 000 (S)	0.9936	4570	0.9994	4050	0.9985
33	3740	147 000	0.9940	4630	0.9994	4140	0.9984
34	3803	140 000	0.9942	4670	0.9994	4210	0.9983
35	3908	106 000 (D)	0.9944	4720	0.9993	4300	0.9982
36	4009	61 000	0.9946	4780	0.9992	4390	0.9980
37	4054	39 400	0.9948	4820	0.9992	4460	0.9980
38	4097	29 700	0.9948	4850	0.9992	4510	0.9980
39	4124	23 000	0.9947	4860	0.9992	4540	0.9981
40	4139	18 600	0.9945	4860	0.9993	4560	0.9982
41	4196	15 900	0.9945	4870	0.9993	4580	0.9982
42	4251	14 100	0.9941	4870	0.9993	4600	0.9983
43	4298	12 800	0.9940	4880	0.9993	4620	0.9983
44	4342	11 800	0.9939	4890	0.9994	4640	0.9984
45	4386	11 000	0.9938	4890	0.9994	4660	0.9984
46	4437	10 400	0.9937	4900	0.9994	4690	0.9984

4850 to 4900. Figure 7 shows the predictions of the delayed S-shaped model and the Gompertz model for each week, specifically illustrating the stability of the delayed S-shaped model during the last weeks of testing.

Using the originally suggested selection method, the delayed S-shaped model's predictions should be followed during the test period, since these are higher than the Gompertz model's. In week 46 the difference between actual number of failures detected and the estimate is rather low, and it may not be expected to detect many more failures. After week 46 another 4802 - 4437 = 365 failures were reported. This gives the delayed S-shaped model's final prediction a relative error of 0.020 and the relative error of the Gompertz model is -0.023, see Table 12. Thus, the delayed S-shaped model overestimated, while the Gompertz model underestimated.

If the stability threshold would had been set to 10% of last weeks prediction, as is the case in the original study by Stringfellow and

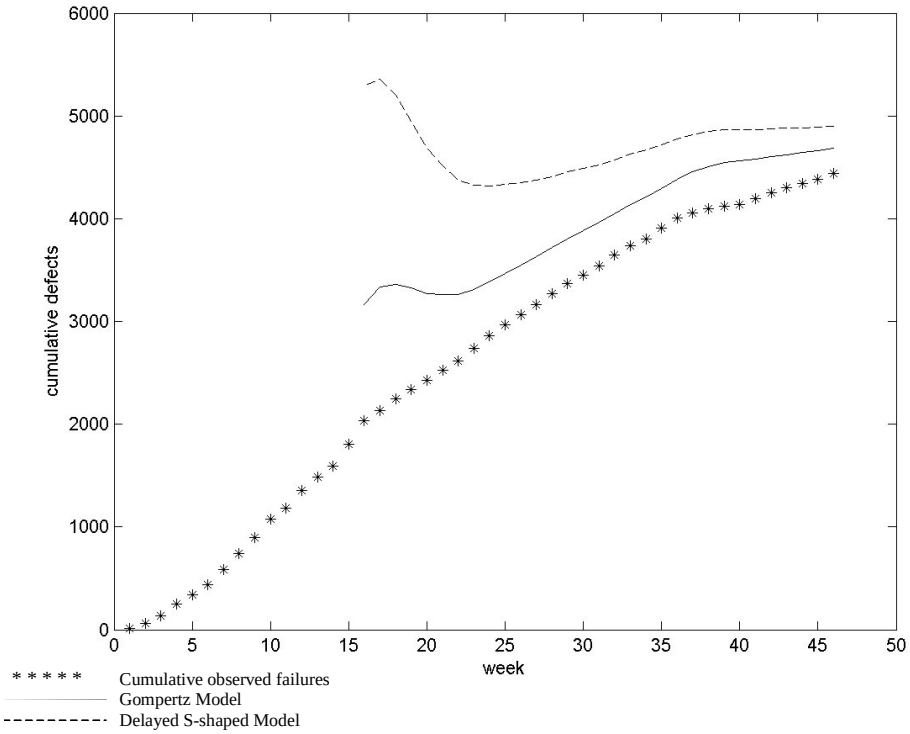

Figure 7. *Plot of project 2 data (FT) and each week's prediction of total number of failures, from SRGMs not rejected.*

Andrews, also the G-O model would have been considered appropriate for the data from function test in project 2. Following this larger stability threshold would imply that the G-O model's estimate is used for predicting the expected number of failures to detect. Consequently, testing would probably have continued another week, although it is unlikely the model would give estimates close to the actual value of total number of failures the nearest weeks. Also, by visual inspection of the cumulative failure curve in Figure 7, it is indicated that an s-shaped model would be more appropriate than a concave model. Thereby, the rejection of the G-O model is justified for the data from function test in project 2, in this case based on the stability threshold.

Table 12. Final estimates and error by SRGMs not rejected for project 2(FT).

Model	**Estimate (true value: 4802)**	**R-value**	**Error**	**Relative error**
Delayed S-shaped	4900	0.9994	98	0.020
Gompertz	4690	0.9984	-112	-0.023

Project 3's failure data from function test and the estimates from the SRGMs are presented in Table 13. Similar to the project 1 and 2, the cumulative failure data are more s-shaped than concave. Thus, the G-O model and the Yamada model were rejected. The G-O model was rejected because a low R-value in week 27, clearly below our threshold on 0.99, although above the threshold value chosen in the original study by Stringfellow and Andrews. In addition, the model destabilizes in week 37 and does not stabilize again, neither according to our stability threshold on 5% of last weeks prediction or the level chosen by Stringfellow and Andrews on 10% of last weeks prediction.

The s-shaped models remain in the selection procedure and both stabilize in week 28. Again, the delayed S-shaped model gives the highest estimate in week 28, and the original selection method tells us that decisions should be based on this model's prediction. Figure 8 illustrates the stability of the estimates from the delayed S-shaped model. Meanwhile the Gompertz model also has a good fit according to the high R-values, but is adjusting its estimates each week to meet the changes in the failure data. Unfortunately, this behaviour could lead to misjudgment, letting management believe that the remaining number of failures is lower

Table 13. Predicted total number of failures for project 2(ST).

Test week FT	Failures found	G-O		Delayed S-shaped		Gompertz	
		Estimate	R-value	Estimate	R-value	Estimate	R-value
27	3403	148 000	0.9847 (R)	5090	0.9976	3880	0.9979
28	3510	145 000 (S)	0.9861	5100 (S)	0.9978	3980 (S)	0.9979
29	3632	157 000 (D)	0.9873	5140	0.9980	4090	0.9978
30	3754	150 000 (S)	0.9884	5180	0.9981	4200	0.9977
31	3900	151 000	0.9893	5270	0.9982	4340	0.9975
32	3954	152 000	0.9902	5300	0.9983	4440	0.9975
33	3993	159 000	0.9908	5310	0.9984	4500	0.9976
34	4054	150 000 (D)	0.9912	5300	0.9985	4550	0.9977
35	4128	143 000 (S)	0.9914	5290	0.9986	4590	0.9978
36	4207	129 000	0.9916	5290	0.9987	4640	0.9978
37	4309	150 000 (D)	0.9918	5310	0.9987	4700	0.9979
38	4385	65 600	0.9919	5330	0.9988	4760	0.9979
39	4447	40 600	0.9919	5350	0.9989	4810	0.9979
40	4511	30 200	0.9920	5360	0.9989	4850	0.9979
41	4552	23 600	0.9920	5370	0.9990	4890	0.9979

than it is in reality. Anyway, this issue is avoided by having a set of models to use in the selection procedure, and in this case the alternative of the delayed S-shaped model is favored, partly because of its higher estimate but also its stable behaviour.

After week 41 and ship date, a rather large amount of failures were reported. However, by following the suggested delayed S-shaped model, this could be predicted. The Gompertz model underestimated the number of failures once again, see Table 14.

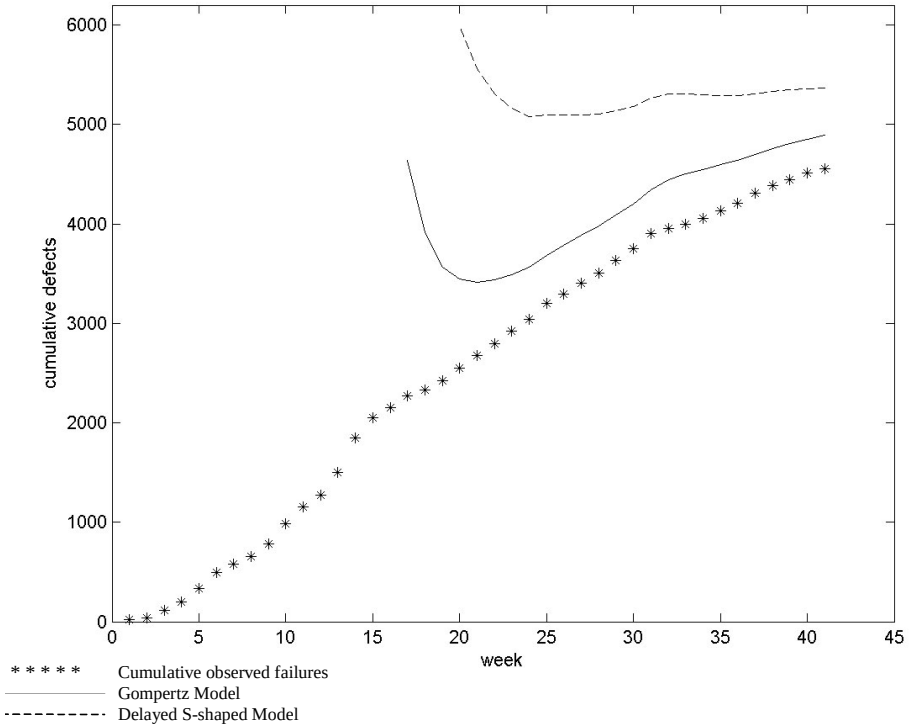

Figure 8. *Plot of project 3 data (FT) and each week's prediction of total number of failures, from SRGMs not rejected.*

Table 14. Final estimates and error by SRGMs not rejected for project 3(FT).

Model	Estimate (true value: 5343)	R-value	Error	Relative error
Delayed S-shaped	5370	0.9990	27	0.005
Gompertz	4890	0.9979	-453	-0.085

The application of SRGMs worked well on the failure data obtained from function testing. The suggested method for selecting model was possible to use also on the data sets consisting of function test failures. It might not be appropriate as support for making stop test decision, although it could be useful as a guide on how many more function test failures to expect. The predictions of the total number of failures for all three projects are all very close to the actual number detected. The

cumulative failure curves for the three projects were s-shaped, and as expected the s-shaped models performed better than the concave. The delayed S-shaped model and the Gompertz model both give reasonable estimates, however, the delayed S-shaped model was considered to be the most appropriate in all cases. The Gompertz model underestimated in all three cases. The concave G-O model was not considered to be appropriate in any case, and did also give very high estimates. The Yamada model did not converge for any of the data set in the beginning of the selection method process, indicating that a good curve fit is not obtainable. For the six data sets in this study, the Yamada model did at several occasions not converge, and required considerable effort to use because of its sensitivity to start point values in the fitting process. Also, the model, when converging, gave predictions with wide confidence limits, which made the practical use of the model more difficult.

Table 15. Relative errors for the SRGMs not rejected for the original study and the replication study. * indicates which model was selected by the selection method.

	Data set	G-O	Delayed S-shaped	Gompertz	Yamada
Original study	Release 1	-	-0.022*	-0.165	-
	Release 2	-	-	-	0.016*
	Release 3	-	0*	-0.036	-
Replication study	Project 1 ST	-	-0.031*	-0.373	-
	Project 2 ST	0.744*	-0.113	-0.128	0.743
	Project 3 ST	0.318*	-0.120	-0.086	0.318*
	Project 1 FT	-	-0.002*	-0.234	-
	Project 2 FT	-	0.020*	-0.023	-
	Project 3 FT	-	0.005*	-0.085	-

3.5 Findings compared to the original study

Stringfellow and Andrews concluded that the selection method worked well when applied on the empirical data they had. Our data differ from their with respect to development environment and software system domain. In addition, our data consists of more data points, both in terms of number of failures and test weeks. Similar to Stringfellow and Andrews, our results show that when applying the selection method to system test data, at least one model is applicable for the data set by the time testing

was approaching a decision point for stopping. Also, our modified calendar time, adjusted with regard to vacations and regression periods, did not provide any obstacles, the models gave reasonable estimates anyway.

Comparable to the original study, we did obtain some diversification among the selected models. By following the conservative choice the highest estimate, the delayed S-shaped model was considered most appropriate for the data set of system test failures from project 1, and the concave model were considered appropriate for system test failures from project 2 and 3. However, in the latter two cases a different selection criterion would be suitable, since the concave models' high estimate resulted in rather high relative error. In a situation like this, the selection method criterion could not be followed strictly by the rules. Each accepted model's estimate must be evaluated to find out whether it is realistic and might be considered reliable for making a stop test decision or not. Generally, the s-shaped models performed better than the concave models in predicting the total number of failures, and they had rather low relative error, often ranging in the small amount of a couple of percents, see Table 15. However, the s-shaped models underestimated in each prediction of the system test failures, even though the absolute value of the relative error is lower than for the concave models. As seen in Table 15, the Gompertz model rather often has a low relative error when applied to the six data sets in this study, but is in each occasion underestimating the number of failures. For the two data sets in the study by Stringfellow and Andrews where the Gompertz model was accepted, the model also underestimates more than the delayed S-shaped model, although the GOF measure was higher than for the delayed S-shaped model.

Compared to the original study, we changed the threshold values for GOF and stability. The GOF R-value was set to 0.99 instead of 0.95. The change manifested itself once, in project 3 function test failures. If the G-O model had been accepted, the model's very high estimate would have an impact on the assessment, since the estimate is very much higher than the other non-rejected models. On the other hand, the G-O model was rejected anyway since it did not stabilize for the data set from project 3 function test failures, leaving the question about the high estimate unnecessary to address. The stability threshold was set to 5% of last week's estimate, instead of 10%. With the higher threshold value, generally the models would stabilize earlier, and provide estimates that

could give more confidence. For the data sets in this study, e.g. the concave models would stabilize earlier, and affect the choice of which estimate to follow in the selection method. This phenomenon is not entirely eligible, shown by the results where the s-shaped models obviously provide estimates with lower relative error. In one occasion, function test failures from project 2, the G-O model is rejected based on our lower threshold value for stability. With the threshold value from the original study, the model would have been accepted instead and would have had impact on the choice of which estimate to follow. The high estimate from the G-O model would probably confuse the prediction of the total number of function test failures to expect, having to consider this value compared to the s-shaped models' estimates. On their own, the s-shaped models were shown to perform rather well in their predictions.

The results of applying the SRGMs on the data sets consisting of function test failures were unexpectedly positive. Despite the fact that by applying the SRGMs to function test data instead of system test data, basic assumptions of the SRGMs are violated, the application of the models could provide good predictions of how many failures to expect in function test. The three data sets were s-shaped and thereby the two s-shaped models fitted very well and resulted in low relative error, while the concave models did not fit very well at all.

4. Conclusions

In this study, we have replicated a selection method for software reliability growth models, and investigated the method by application to empirical software failure data. By replicating the original study and comparing the results of the selection method across various types of projects, more confidence in the results are obtained. Hence, the replication study is one step towards generalization of the procedure. In the replication study, the main selection method has been held constant, while other parameters have been varied, compared to the original study. Parameters such as the threshold values for the evaluation criteria have been changed. In addition, the failure data are obtained from a different development environment. The selection method may be further evaluated by varying other parameters than the ones discussed in this study, to gain even more confidence in the selection method. One suggestion is to apply the selection method with a different set of software reliability growth

models. The decision of which parameters to change, when applying the selection method, has to be based on the environment where the application occurs. The method has been shown to perform its best while adapted to the circumstances that are specific to each development environment.

Evidently, other selection methods for software reliability models could be used. Fenton and Pfleeger [3] present several procedures for comparing different models, such as examining the basic assumptions given for each model, degree of bias, and prequential likelihood. The selection method suggested by Stringfellow and Andrews consists of several well-defined steps, building up a complete procedure to follow. Its direct applicability to an industrial case motivated our choice of further investigating this selection method.

The results of the replication confirm that the basic ideas of the selection method works well. Generally, the failure detection pattern converges toward values predicted by the models. This projection allows management to stop testing at an appropriate time with confidence in the quality of the developed software. Nevertheless, the selection of which model to base the stop testing decision upon is not always in accordance with the choice suggested by the original study. Indicated by the data sets in this study, by following the conservative choice of trusting the model giving the highest estimate, unnecessary amount of testing might be spent. This replication study thus suggests that the model selection and thereby conclusions from Stringfellow and Andrews to some extent will depend on the chosen values for stability and GOF, and when more than one stable model, the maximum estimate. As shown in this study, these mentioned criteria might need to be changed, to support the selection method in an appropriate way and make it applicable also in other development environments. Besides that, the selection method works as an appropriate tool for selecting SRGMs in a systematic way often resulting in predictions with a low relative error when compared to the actual values of total number of failures. Nevertheless, also mentioned by Stringfellow and Andrews, the selection method should better be used with complementary techniques for assessment of the developed software [19], and not be trusted as the single-handed method.

In addition to applying the selection method and its SRGMs on system test data, the procedure was tried out on function test data. The models provided good predictions of the total number of function test failures to expect to find. The results show that the application of SRGMs

could be valuable also to other types of data, even though several of the underlying assumptions of the models are violated.

Acknowledgement. The author would like to thank Dr. Catherine Stringfellow and Prof. Anneliese Andrews for being generous with their time and willing to answer my questions about their selection method. Thanks also to Prof. Per Runeson who provided valuable comments on the paper.

5. References

[1] Ehrlich, W., Lee, S. and Molisanim R., "Applying reliability measurement: a case study", *IEEE Software*, 7(2): 56-64, 1990.

[2] Ehrlich, W., Prasanna, B., Stampfel, J. and Wu, J., "Determining the cost of a stop-test decision", *IEEE Software*, 10(2): 33-42, 1993.

[3] Fenton, N. E. and Pfleeger, S. L., *Software Metrics: A Rigorous and Practical Approach* (2nd ed.), Boston: PWS Publishing Company, 1997.

[4] Fujiwara, T. and Yamada, S., "A testing-domain-dependent software reliability growth model for imperfect debugging environment and its evaluation of goodness-of-fit", *Electronics and Communications in Japan*, Part 3 86(1): 11-18, 2003.

[5] Gaudoin, O., Yang, B. and Xie, M., "A Simple Goodness-of-Fit Test for the Power-Law Process, Based on the Duane Plot", *IEEE Transaction on Reliability*, 52(1): 69-74, 2003.

[6] Goel, A. L., and Okumoto, L., "A time dependent error detection rate model for software reliability and other performance measures", *IEEE Transactions on Reliability*, 28(3): 206-211, 1979.

[7] Institute of Electrical and Electronics Engineers, *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std 610.12-1990, 1990.

[8] International Standards Organisation, *Information Technology - Software Product Evaluation - Quality Characteristics and Guide Lines for their Use*, ISO/IEC FDIS 9126-1. Geneva, Switzerland, 2000.

[9] Jeske, D. R., and Zhang, X., "Some successful approaches to software reliability modeling in industry", *The journal of Systems and Software*, 74(1): 85-99, 2005.

[10] Kececioglu, D., *Reliability Engineering Handbook,* Vol. 2. Englewood Cliffs, NJ: Prentice-Hall, 1991.

[11] Lyu, M. R. (ed.)., *Handbook of Software Reliability Engineering*, New York: McGraw-Hill, 1996.

[12] Miller, J., "Replicating software engineering experiments: a poisoned chalice or the Holy Grail", *Information and Software Technology*, 47(4): 233-244, 2005.

[13] Montgomery, D. C., *Design and Analysis of Experiments* (5th ed.), New York: John Wiley & Sons, 2001.

[14] Musa, J., Iannino, A. and Okumoto, L., *Software Reliability Measurement, Prediction, Application*, New York: McGgraw-Hill, 1987.

[15] Musa, J. and Ackerman, A., "Quantifying software validation: when to stop testing?", *IEEE Software*, 6(3): 19-27, 1989.
[16] Musa, J., *Software Reliability Engineering,* New York: McGraw-Hill, 1999.
[17] Robson, C., *Real World Research*, UK: Blackwell Publishers, 2002.
[18] Siegel, S. and Castellan, N. J., *Nonparametric Statistics for the Behavioral Sciences*, Singapore: McGraw-Hill, 1988.
[19] Stringfellow, C., *An integrated method for improving testing effectiveness and efficiency*, PhD Dissertation, Colorado State University, 2000.
[20] Stringfellow, C. and Amschler Andrews, A., "An Empirical Method for Selecting Software Reliability Growth Models", *Empirical Software Engineering*, 7(4): 319-343, 2002.
[21] Wood, A., "Predicting software reliability", *IEEE Computer*, 29(11): 69-78, 1996.
[22] Wood, A., "Software reliability growth models: Assumptions vs. reality", *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pp. 136-141, 1997.
[23] Yamada, S., Ohba, M. and Osaki, S., "S-shaped reliability growth modeling for software error detection", *IEEE Transaction on Reliability,* 32(5): 475-478, 1983.
[24] Yamada, S., Ohtera, H. and Narihisa, H., "Software reliability growth models with testing effort", *IEEE Transactions on Reliability,* 35(1): 19-23, 1986.

Reports on Communication Systems

101 **On Overload Control of SPC-systems**
Ulf Körner, Bengt Wallström, and Christian Nyberg, 1989.

102 **Two Short Papers on Overload Control of Switching Nodes**
Christian Nyberg, Ulf Körner, and Bengt Wallström, 1990.

103 **Priorities in Circuit Switched Networks**
Åke Arvidsson, *Ph.D. thesis*, 1990.

104 **Estimations of Software Fault Content for Telecommunication Systems**
Bo Lennselius, *Lic. thesis*, 1990.

105 **Reusability of Software in Telecommunication Systems**
Anders Sixtensson, *Lic. thesis*, 1990.

106 **Software Reliability and Performance Modelling for Telecommunication Systems**
Claes Wohlin, *Ph.D. thesis*, 1991.

107 **Service Protection and Overflow in Circuit Switched Networks**
Lars Reneby, *Ph.D. thesis*, 1991.

108 **Queueing Models of the Window Flow Control Mechanism**
Lars Falk, *Lic. thesis*, 1991.

109 **On Efficiency and Optimality in Overload Control of SPC Systems**
Tobias Rydén, *Lic. thesis,* 1991.

110 **Enhancements of Communication Resources**
Johan M. Karlsson, *Ph.D. thesis*, 1992.

111 **On Overload Control in Telecommunication Systems**
Christian Nyberg, *Ph.D. thesis*, 1992.

112 **Black Box Specification Language for Software Systems**
Henrik Cosmo, *Lic. thesis*, 1994.

113 **Queueing Models of Window Flow Control and DQDB Analysis**
Lars Falk, *Ph.D. thesis*, 1995.

114 **End to End Transport Protocols over ATM**
Thomas Holmström, *Lic. thesis*, 1995.

115 **An Efficient Analysis of Service Interactions in Telecommunications**
Kristoffer Kimbler, *Lic. thesis*, 1995.

116 **Usage Specifications for Certification of Software Reliability**
Per Runeson, *Lic. thesis,* May 1996.

117 **Achieving an Early Software Reliability Estimate**
Anders Wesslén, *Lic. thesis*, May 1996.

118 **On Overload Control in Intelligent Networks**
Maria Kihl, *Lic. thesis,* June 1996.

119 **Overload Control in Distributed-Memory Systems**
Ulf Ahlfors, *Lic. thesis,* June 1996.

120 **Hierarchical Use Case Modelling for Requirements Engineering**
Björn Regnell, *Lic. thesis,* September 1996.

121 **Performance Analysis and Optimization via Simulation**
Anders Svensson, *Ph.D. thesis*, September 1996.

122 **On Network Oriented Overload Control in Intelligent Networks**
Lars Angelin, *Lic. thesis,* October 1996.

123 **Network Oriented Load Control in Intelligent Networks Based on Optimal Decisions**
Stefan Pettersson, *Lic. thesis,* October 1996.

124 **Impact Analysis in Software Process Improvement**
Martin Höst, *Lic. thesis,* December 1996.

125 **Towards Local Certifiability in Software Design**
Peter Molin, *Lic. thesis,* February 1997.

126 **Models for Estimation of Software Faults and Failures in Inspection and Test**
Per Runeson, *Ph.D. thesis*, January 1998.

127 **Reactive Congestion Control in ATM Networks**
Per Johansson, *Lic. thesis,* January 1998.

128 **Switch Performance and Mobility Aspects in ATM Networks**
Daniel Søbirk, *Lic. thesis,* June 1998.

129 **VPC Management in ATM Networks**
Sven-Olof Larsson, *Lic. thesis,* June 1998.

130 **On TCP/IP Traffic Modeling**
Pär Karlsson, *Lic. thesis,* February 1999.

131 **Overload Control Strategies for Distributed Communication Networks**
Maria Kihl, *Ph.D. thesis*, March 1999.

132 **Requirements Engineering with Use Cases – a Basis for Software Development**
Björn Regnell, *Ph.D. thesis*, April 1999.

133 **Utilisation of Historical Data for Controlling and Improving Software Development**
Magnus C. Ohlsson, *Lic. thesis,* May 1999.

134 **Early Evaluation of Software Process Change Proposals**
Martin Höst, *Ph.D. thesis*, June 1999.

135 **Improving Software Quality through Understanding and Early Estimations**
Anders Wesslén, *Ph.D. thesis*, June 1999.

136 **Performance Analysis of Bluetooth**
Niklas Johansson, *Lic. thesis,* March 2000.

137 **Controlling Software Quality through Inspections and Fault Content Estimations**
Thomas Thelin, *Lic. thesis,* May 2000.

138 **On Fault Content Estimations Applied to Software Inspections and Testing**
Håkan Petersson, *Lic. thesis,* May 2000.

139 **Modeling and Evaluation of Internet Applications**
Ajit K. Jena, *Lic. thesis,* June 2000.

140 **Dynamic traffic Control in Multiservice Networks – Applications of Decision Models**
Ulf Ahlfors, *Ph.D. thesis*, October 2000.

141 **ATM Networks Performance – Charging and Wireless Protocols**
Torgny Holmberg, *Lic. thesis,* October 2000.

142 **Improving Product Quality through Effective Validation Methods**
Tomas Berling, *Lic. thesis,* December 2000.

143 **Controlling Fault-Prone Components for Software Evalution**
Magnus C. Ohlsson, *Ph.D. thesis*, June 2001.

144 **Performance of Distributed Information Systems**
Niklas Widell, *Lic. thesis,* February 2002.

145 **Quality Improvement in Software Platform Development**
Enrico Johansson, *Lic. thesis,* April 2002.

146 **Elicitation and Management of User Requirements in Market-Driven Software Development**
Johan Natt och Dag, *Lic. thesis,* June 2002.

147 **Supporting Software Inspections through Fault Content Estimation and Effectiveness Analysis**
Håkan Petersson, *Ph.D. thesis,* September 2002.

148 **Empirical Evaluations of Usage-Based Reading and Fault Content Estimation for Software Inspections**
Thomas Thelin, *Ph.D. thesis,* September 2002.

149 **Software Information Management in Requirements and Test Documentation**
Thomas Olsson, *Lic. thesis,* October 2002.

150 **Increasing Involvement and Acceptance in Software Process Improvement**
Daniel Karlström, *Lic. thesis,* November 2002.

151 **Changes to Processes and Architectures; Suggested, Implemented and Analyzed from a Project viewpoint**
Josef Nedstam, *Lic. thesis,* November 2002.

152 **Resource Management in Cellular Networks -Handover Prioritization and Load Balancing Procedures**
Roland Zander, *Lic. thesis,* March 2003.

153 **On Optimisation of Fair and Robust Backbone Networks**
Pål Nilsson, *Lic. thesis,* October 2003.

154 **Exploring the Software Verification and Validation Process with Focus on Efficient Fault Detection**
Carina Andersson, *Lic. thesis,* November 2003.

155 **Improving Requirements Selection Quality in Market-Driven Software Development**
Lena Karlsson, *Lic. thesis*, November 2003.

156 **Fair Scheduling and Resource Allocation in Packet Based Radio Access Networks**
Torgny Holmberg, *Ph.D. thesis*, November 2003.

157 **Increasing Product Quality by Verification and Validation Improvements in an Industrial Setting**
Tomas Berling, *Ph.D. thesis*, December 2003.

158 **Some Topics in Web Performance Analysis**
Jianhua Cao, *Lic. thesis*, June 2004.

159 **Overload Control and Performance Evaluation in a Parlay/OSA Environment**
Jens K. Andersson, *Lic. thesis*, August 2004.

160 **Performance Modeling and Control of Web Servers**
Mikael Andersson, *Lic. thesis*, September 2004.

161 **Integrating Management and Engineering Processes in Software Product Development**
Daniel Karlström, *Ph.D. thesis*, December 2004.

162 **Managing Natural Language Requirements in Large-Scale Software Development**
Johan Natt och Dag, *Ph.D. thesis*, Febuary 2005.

163 **Designing Resilient and Fair Multi-layer Telecommunication Networks**
Eligijus Kubilinskas, *Lic. thesis*, Febuary 2005.

164 **Internet Access and Performance in Ad hoc Networks**
Anders Nilsson, *Lic. thesis*, April 2005.

165 **Active Resource Management in Middleware and Service-oriented Architectures**
Niklas Widell, *Ph.D. thesis*, May 2005.

166 **Quality Improvement with Focus on Performance in Software Platform Development**
Enrico Johansson, *Ph.D. thesis*, June 2005.

167 **On Inter-System Handover in a Wireless Hierarchical Structure**
Henrik Persson, *Lic. thesis*, September 2005.

168 **Prioritization Procedures for Resource Management in Cellular Networks**
Roland Zander, *Ph.D. thesis*, December 2005.

169 **Strategies for Management of Architectural Change and Evolution**
Josef Nedstam, *Ph.D. thesis*, December 2005.

170 **Internet Access and QoS in Ad Hoc Networks**
Ali Hamidian, *Lic. thesis*, April 2006.

171 **Managing Software Quality through Empirical Analysis of Fault Detection**
Carina Andersson, *Ph.D. thesis*, May 2006.