Exploring the Software Verification and Validation Process with Focus on Efficient Fault Detection

Andersson, Carina

2003

Exploring the Software Verification and Validation Process with Focus on Efficient Fault Detection

Carina Andersson

LUND UNIVERSITY

Department of Communication Systems
Lund Institute of Technology

ISSN 1101-3931
ISRN LUTEDX/TETS--1062--SE+134P

Printed in Sweden
KFS AB
Lund 2003

This thesis is submitted to Research Board FIME – Physics, Informatics, Mathematics and Electrical Engineering – at Lund Institute of Technology (LTH), Lund University, in partial fulfilment of the requirements for the degree of Licentiate of Technology in Software 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

Abstract

Quality is an aspect of high importance in software development projects. The software organizations have to ensure that the quality of their developed products is what the customers expect. Thus, the organizations have to verify that the product is functioning as expected and validate that the product is what the customers expect. Empirical studies have shown that in many software development projects as much as half of the projected schedule is spent on the verification and validation activities.

The research in this thesis focuses on exploring the state of practice of the verification and validation process and investigating methods for achieving efficient fault detection during the software development. The thesis aims at increasing the understanding of the activities conducted to verify and validate the software products, by the means of empirical research in the software engineering domain.

A survey of eleven Swedish software development organizations investigates the current state of practice of the verification and validation activities, and how these activities are managed today. The need for communicating and visualising the verification and validation process was expressed during the survey. Therefore the usefulness of process simulations was evaluated in the thesis. The simulations increased the understanding of the relationships between different activities among the involved participants. In addition, an experiment was conducted to compare the performance of the two verification and validation activities, inspection and testing.

In the future work, empirical research, including experiment results, will be used for calibration and validation of simulation models, with focus on using simulation as a method for decision support in the verification and validation process.

Acknowledgements

This work was partly funded by VINNOVA under a grant for LUCAS – the Center of Applied Software Research at Lund University.

Firstly, I would like to thank my supervisor, Per Runeson, for giving me an opportunity to do my postgraduate studies in the Software Engineering Research Group and for his guidance during these first years. I would also like to thank my assistant supervisor, Thomas Thelin, for his support and patience.

I am grateful to the co-authors of my papers and others who have contributed to the research in this thesis.

I would like to thank my colleagues in the Software Engineering Research Group, for an inspiring and supporting atmosphere.

I would also like to mention the colleagues at the Department of Communication Systems, thanks for providing an excellent environment to work in, and interesting, though sometimes endless, conversations during the coffee breaks.

My thanks and hugs to my friends for giving me a joyful time when not working. To "the girls", what should I do without your crazy ideas for how to spend a weekend? My life would be much more boring without our events, even if these also cause both sweat and pain, and more often than not: soaking clothes because of pouring rain. To the "horse friends" for two reasons, our common interest that has grown to include several more, and for challenging me to improve in an area outside the professional. To Jenny and Louise, for listening even when I'm not in the best of moods.

I would like to thank my family, mum and dad for always being there for me, my sisters, Eva and Irene for looking after me and never letting me forget that there are different perspectives of life, and my brother Jan, without you I never would had started my career in the technical domain.

Finally, my thanks to Jonas, for being my most devoted supporter during the last two years.

Contents

Introduction

Today, software plays an important role in modern technology. A substantial proportion of all products, both commercial products and other application domains, contain some kind of software. And the trend is that each product contains more and more software every year.

Independently of the field of application, one major factor is in focus when using the software: the quality of the software product. With a product quality below expectations the customers will shortly find a substitute product that better satisfies his or her needs. Therefore, the software development organizations are forced to assure that their products are of acceptable quality, though not exceeding the budget and still delivered on the agreed schedule. In the 60s, the so-called software crisis was identified. The fact that the developed software was of low quality, over budget, and behind schedule was considered in almost every article on software engineering. This viewpoint has changed, saying the practice in software engineering is doing pretty well [11]. Compared to other engineering disciplines, e.g. the building trade, whose projects also happens to be late and over budget, the software engineering discipline is producing rather good results, in spite of the fact that this discipline is much younger. However, there is still much to improve in the software engineering field. As the software market expands and the customers demand more complex systems, the problems still exist [10], [41]. Thus, in the software engineering discipline, as well as in many other

engineering disciplines, there are much to gain by further research and improvements.

Of the three objectives, software quality, costs according to budget, and delivery on time, this thesis concentrates on the first: software quality. The quality control and assurance activities impel the organizations to examine the developed artefact to ensure that it meets the desired quality. Once the product is constructed, the quality has to be evaluated, i.e. the organizations have to validate that the product is what the customers requested, and verify that the product functions as expected.

The triangle in Figure 1, which the three objectives creates, with quality, cost, and schedule, in each corner, visualizes that each objective is dependent on the others. This thesis focuses on the importance of the verification and validation process, and aims to increase the understanding of the practices and activities that may be chosen to examine the developed software artefacts and assure the product quality. However, as the triangle illustrates, costs and schedule cannot be excluded from the context, which also is shown in the research reported in this thesis. The interest in a verification and validation activity is always combined with the interest in its cost and consumption of time.

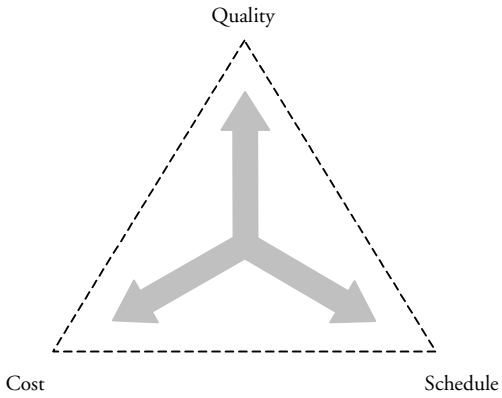


Figure 1. *The three objectives in software engineering.*

The first part of this thesis is an introductory part that summarises the work. The introduction is organised as follows: in Section 1, the different concepts addressed in this thesis are described. Furthermore, the research focus and the purpose of the thesis are discussed together with the

research questions. In Section 2, work related to the research in the thesis is summarised. In Section 3, the practised research methodology is discussed. The research results and the main contributions of this thesis are discussed in Section 4, together with the threats to validity. In Section 5, the plan for the future research is presented, impelled by the results of this thesis.

The following papers are included in the thesis.

PAPER 1. 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.

PAPER 2. 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.

PAPER 3. 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 4. 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.

In addition to the papers mentioned above, the author has contributed to the following papers:

Understanding Software Processes through System Dynamics Simulation: A Case Study

Carina Andersson, Lena Karlsson, Josef Nedstam, Martin Höst, Bertil I Nilsson

Proceedings of 1st Swedish Conference on Software Engineering Research and Practice, pp. 1-8, 2001. This paper is an earlier version of paper 2.

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.

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

To appear in Empirical Software Engineering: An International Journal, 2003. This is an extended version of the paper above.

1. Research Focus

The research presented in this thesis is in the field of software engineering. Software development does not solely consist of programming or implementing code modules. Several more activities are necessary to produce high-quality, maintainable software products. The research in this thesis more specifically concerns the verification and validation activities in the software development process.

A widely used representation of the practical development is the waterfall model [34]. The fundamental development activities in the waterfall model are represented by succeeding process phases, and in principle is the result of each phase one or more artefacts. The straightforward approach is useful in large software projects with well defined requirements. However, the model is too inflexible, with distinct phases, when the requirements are changeable. The basic idea of the waterfall model is still used, though it has been refined to suit e.g. incremental development, to support parallel activities.

Verification and validation are important parts of the software development process. The verification and validation activities ensure that the artefacts conform to the specifications and that the final product is what was initially requested from the customers. This may start from the first development phase, with analysis of the requirements, and proceed during the development with reviews of design documents and code and product testing. Ideally, the faults in a software product are discovered as early as possible, to avoid expensive rework later on in the development process. Costs to detect and correct faults grow dramatically when these have propagated to later phases. Observations show that the costs of correcting a fault in design and code phases is often 10 to 100 times less expensive than if it is found during the test phases [1]. This is a major driver for focusing the effort on detecting faults early in the development process.

The verification and validation activities can be divided into two different strategies, static and dynamic [38]. Static techniques do not require that the system is executed and may be applied at all stages of the development process. Static techniques may be applied as formal reviews like inspections [9], or automatic analyses of the code of a system or associated documents. However, to ensure that a program is operationally correct also dynamic techniques, which mean that the system is executed with test data, are necessary [13]. On the other hand, testing is only possible when a prototype or an executable version of a program is available.

Both static and dynamic techniques aim at detecting software faults[1], but it is not evident which faults are best found by which technique. Still, faults can be injected into the software products at all stages in the

1. In this thesis a fault is defined as an incorrect step, process, or data definition included in any of the developed products. A failure is an incorrect result during execution, i.e. occurs when a software system does not behave as desired, and thereby reveals a fault [14].

development process, and can also through the verification and validation activities be removed from these. Table 1 shows examples of activities in the development process in which faults are injected and removed.

To be most effective, the verification and validation activities must be fully integrated in the development process. The V-model [34] is used for structuring the software development process with several detailed steps, guiding the developer through the software engineering process. The

Table 1. Activities associated with fault injection and fault removal. Derived from [18].

Development phase	Fault injection	Fault removal
Requirements	Requirements gathering process and the development of specifications	Requirements verification
Design	Design work	Design verification
Code implementation	Coding	Code verification
Integration	Integration process	Integration test
Unit test	Bad fixes	Testing itself
System test	Bad fixes	Testing itself

name origins from the main shape of the model, which is designed as a V. Figure 2 shows an example of a development process with its generic products and activities as well as the corresponding verification and validation activities. The left hand side of the model can be seen as the waterfall model from requirements engineering, design definition, down to implementation of program code [38], including subsequent verification activities. The right hand side describes the testing activities, unit tests, integration tests, system tests, and acceptance tests.

However, the V-model could also be considered as a product model, rather than a process model. This since the model not necessarily defines the sequence of the development steps in which the products are created. Hence, the model is applicable to different types of development processes, with parts developed in sequence, parallel, or incrementally.

An important aspect in the model is the relations between the artefacts and the corresponding quality control activities, represented by the dotted lines in Figure 2. The model expresses that the requirements have to be consistent with the results of the corresponding tests, the high and low-

level design have to be consistent with their corresponding tests. Another purpose with the model is that the relations between the developed products should be maintained and an inspection can be conducted as soon as an artefact, or part of an artefact, is created. A typical example is inspection of code that is checked against the design document.

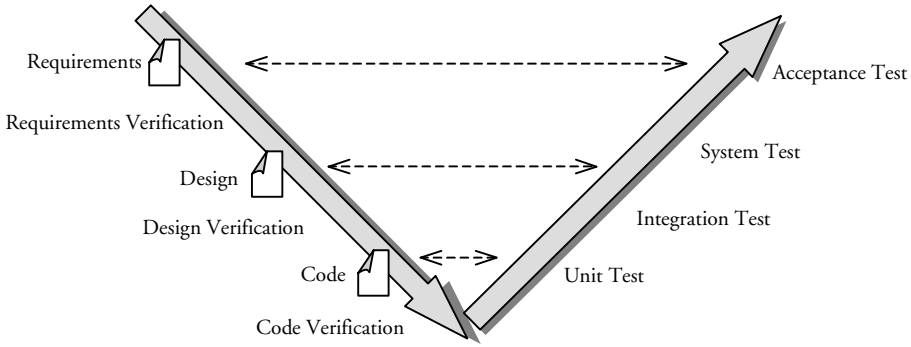


Figure 2. *V-model.*

1.1 Terminology

Among the inspection and testing activities a variety of methods and techniques exists. This section describes the techniques that are applied in the research discussed in the following sections of the thesis.

Inspections [9] can be more or less formal. Walkthroughs are less formal, with no additional effort for the participants than the main inspection meeting, while more formal inspections require preparation before the meeting. During the inspection the participants are provided with guidelines, e.g. checklists, to assist the participants in the task of finding faults. Other inspection techniques have been proposed, e.g. reading techniques that focus on the inspected artefacts from different perspectives [5] where the participants more actively produce some result. Reading by stepwise abstraction is an example of an active reading technique [24].

The testing techniques discussed in the thesis are either functional testing or structural testing. Functional testing is based on the input and the output of the program under test, without any internal knowledge of

the program. Examples of methods for functional testing are equivalence partitioning, where the input are classified into representative sets, and boundary-value analysis, which is a variant of equivalence partitioning with the difference that the tested input are selected from the edges of the sets [19]. The testing can also be termed requirements-based, i.e. the software is tested based on the requirements without knowledge of the low-level design or the code. Structural testing is based on the internal program structure and is derived from the low-level design specification or the code. Examples of methods for structural testing are control flow analysis, e.g. statement coverage, where each statement in the program under test should be executed at least once, and data flow analysis, which focuses on how the data flow through a sequence of processing steps in the program. Design-based testing, which has a design coverage goal, is an example of a data flow method [37].

Another approach than testing is to provide fault tolerance in the developed software. An example in this field is voting, which is used as fault detection mechanism in N-version programming [37], [38]. Self-checks are similar to the acceptance tests for recovery blocks and means that each program component includes a test to check if the component has executed successfully [37], [38].

1.2 Research Purpose

In many software development projects half of the projected schedule is spent on verification and validation activities [8]. Therefore, the research in this thesis aims at creating a better understanding of the verification and validation process. The objective of the research is to explore the current state of the practice, investigate verification and validation techniques, and evaluate methods for visualizing relationships between factors that both affect the process and the quality of the developed product. Thus, the research scope in this thesis is limited to empirical research on the verification and validation process, with a special focus on the activities inspection and testing.

The research focus on exploring the verification and validation process has impelled the research questions listed below. The questions are listed in the order they are addressed in the thesis.

RQ1. How do software organizations perform verification and validation?

RQ2. Can the understanding of the verification and validation process, and its activities, be increased by simulations?

RQ3. How do inspections compare to testing regarding performance in finding design faults?

To form a basis for the research in this thesis, the following section describes related work in this area. Specifically research investigating the verification and validation activities inspection and testing in relation to each other is presented.

2. Related Work

The core questions in much of the empirical software engineering research are investigating objects in terms of one or more of the three objectives, quality, cost, or schedule. If several alternatives for conducting a certain activity are available, which is the best one? If there is a best alternative, can it be standardized to suit different organizations? 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 strategies separately, can be found in [3], which describes the state-of-art of inspections, and in [16], which describes empirical studies on testing.

The following summarises a survey of empirical work investigating inspections and testing, with the scope limited to studies including evaluation of *both* inspections and testing. The survey covers case studies, simulation studies and experiments.

2.1 Case Studies

Several case studies, investigating inspections and testing, are reporting on experiences from industry. These often have a comprehensive viewpoint, though are not presenting any statistically guaranteed results. One observed distinction from e.g. 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 artefacts than code is mentioned. Hence, inspections of documents, like requirements and design specifications are also in focus.

The most addressed question in the case studies investigating inspections and testing is a simplification of 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 [18], [35]. 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].

2.2 Simulation Studies

Simulation is another approach for creating an understanding of the mechanisms affecting the development process. An implemented computer model, calibrated with empirical data, can be executed, simulating the real process behaviour. Empirical issues related to software process simulations concern the analysis of the process data, as direct input to the model or used for the model building, the model output data, used as support for planning and management decisions, and the model structure 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, like the unit test phase. The benefit of simulation modelling is that to get an accurate model the activities have to be clearly identified. Furthermore, dependencies, as well as the feedback between various activities, have to be defined. The following summarized simulation studies focus on inspection and testing. The models are empirically validated with industry data.

Madachy: Madachy developed a dynamic simulation model (see Section 3.2) of a development process with the aim to investigate the inspection activities and their effects on cost, schedule and quality [25]. Inspections and system testing were assumed to be the only fault detection activities in the modelled organization. The data collected from

two projects was used for validation. In addition, data from the literature [7] were used for calibration.

Findings: The simulation results were compared, and validated against project data and other published data. These showed that the extra effort required for inspections in the design and coding phase, is saved by the reduced testing and integration effort. However, for too low fault rates, inspections were not cost effective. The feedback effect of inspections, which tends to reduce the fault generation rate, could also be seen in the executions of the simulation model. The results also discussed the trade-off for performing design inspections, code inspections, or both e.g. while changing the values for the rework cost during testing.

Raffo and Kellner: Raffo and Kellner conducted a field study to evaluate a specific process change [30]. The selected problem was to assess the impact of creating unit test plans during the coding phase. Other considered process changes included high-level and low-level design phases, conducting unit testing before code inspection, and inspecting only high-risk modules of the product at the design and code phase. Hence, the studied process change consisted of a modification of the unit test phase, requiring unit test plans to be developed during the coding phase and also inspected during the code inspection.

Findings: The executions showed that the proposed process change would lead to significant reductions in remaining faults, effort to correct field detected faults, and project duration. They also showed that, at a higher implementation cost, improving the inspections could be a more effective process improvement than the creating unit test plans change.

Martin and Raffo: Martin and Raffo developed a hybrid model (see Section 3.2) of a software development process [26]. The aim of the simulation was to study the effects of having a pool of experienced engineers and eliminating the unit test step. Changing the process to remove the unit test involves a change to the activities composing the process, modelled as discrete events. The higher experience among the engineers will give feedback effects as lower fault injection rates, and increase fault detection, which have to be represented by a system dynamics model. Input data for the model were collected in a real-world process over two years, while project managers estimated some data. Model output variables of interest were the number of undetected faults, and the effect on effort and duration, as results of the changes.

Findings: The results were statistically tested by a two-way ANOVA [27]. To summarize the results, if implementing both changes, the project duration and the effort would be reduced significantly, and the number of undetected faults would decrease.

Development of process simulation models requires determining the accurate input parameters, which could be representative distributions for key project parameters, such as productivity, fault injection 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, it is only simplistic models representing the real world, and the model results are highly dependable on the accuracy of the input. In the literature several suggested simulations models can be found, though they are more seldom applied in a real-world environment, and not validated with project data, since adequate data describing the quantitative relationships are not often available.

2.3 Experimental Studies

Several experiments have investigated and compared the impact of inspections and testing separately. Investigations on inspection, e.g. by comparing different reading techniques [39], and on testing by comparing the effectiveness of different test techniques [31].

The surveyed experiments from the last three decades do all include some form of inspection and testing applied on code modules, compared to each other.

Hetzel: Hetzel's experiment from 1976, used 39 subjects, mostly students with little programming experience, and evaluated three methods: functional testing, a variation of structural testing, and code reading [12].

Findings: The experiment resulted in the same effectiveness for functional testing and structural testing and significantly inferior result for the code reading method.

Myers: The experiment of Myers from 1978 included 59 professionals as subjects and compared three different approaches and variations thereof: functional testing, structural testing, and team-based code walkthroughs/ inspections [28].

Findings: The analysis concluded that the three methods were equally effective in finding errors and that none of the methods, when used alone, was very good. The author's implication was that walkthroughs and inspection is a supplement to testing, and not a replacement. On the other hand, the walkthrough/inspection method was more costly, using more labour, than the other methods. The detection of certain classes of faults varied from technique to technique.

Basili and Selby: This experiment from the mid 1980's also compared three techniques: code reading by stepwise abstraction, functional testing using equivalence partitioning and boundary value analysis, and structural testing using 100 percent statement coverage [4]. 32 professional programmers and 42 advanced students were used as subjects. The techniques were evaluated in three different aspects, fault detection effectiveness, fault detection cost, and classes of faults found, applied on four different programs of different software type.

Findings: In this experiment, it was concluded that each technique's effectiveness depended on the software type. Some evidence showed that code reading detected more faults, though this was dependent on the different subject groups.

Kamsties and Lott: This study from the 1990's contributed with two replications of Basili and Selby's experiment, using the same experimental design with some variations [17]. They changed the programs, language and the associated faults. The first replication contained 27 subjects, the second 15, both using students.

Findings: No statistical significance regarding the fault detection effectiveness of the three techniques was obtained. Regarding cost-effectiveness, functional testing was the best.

Roper et al.: This study reported on a replication of Kamsties and Lott's experiment, using 47 students as subjects [33].

Findings: The results supported Kamsties and Lott's, i.e. no statistically significance regarding fault detection effectiveness of the techniques was obtained. They also concluded, as Basili and Selby did, that the effectiveness depended on the programs, the software type, and also on the type of faults in those programs. The third major finding showed that the techniques were more effective when used in combination with each other than when not.

Laitenberger: Laitenberger reports on a controlled experiment with code inspection and structural testing [22]. This experiment differs from the earlier ones in the sense that the 20 subjects, graduate students, applied the techniques sequentially, as adopted in industry.

Findings: The experiment did not confirm that the two techniques complement each other. On the contrary, faults missed in inspection were often not detected in testing either. The first inspection even hindered the effectiveness of structural testing and showed that the two techniques did not focus on different fault classes.

So et al.: The authors conducted first an experiment comparing different testing techniques on eight program versions and conducted then a follow-up experiment, including inspections on five of the eight program versions with 15 graduate students [37].

Findings: The results show that voting, the used testing technique, which consisted of two components: requirements-based and design-based testing, and inspections detected more faults than the other methods, which were code reading by stepwise abstraction, self-checks, and data-flow analysis. The results also indicate that voting, the testing method and inspections were complementary to each other, i.e. faults detected by one method was to a large portion not detected by the other two methods.

To summarize the surveyed experiments, several of the papers finish with calls for "replication of this study is necessary" and "further work on this topic". Deriving reliable results from one single experiment is not very likely. It requires a large number of studies to provide a definitive answer, no matter which discipline is in focus. Still, several of the conducted experiments are replications of each other. From these and the others the following can be concluded:

- There is no consistent evidence that one fault detection technique is superior to the others.
- In most studies is it suggested that inspections and testing should be applied in combination, though this is depending on the choice of technique.
- There is no evidence that one technique is able to detect all existing faults in an artefact exposed to the detection techniques.

2.4 Conclusions from Related Work

Inspections and testing, as the two most important verification and validation techniques, are activities that both researchers and practitioners need to understand, on how to use them separately, but more importantly how to combine them to achieve highest effectiveness.

The survey of the empirical studies comparing inspections and testing, does not give a simple answer of how the techniques are related and how to combine them most effectively. On the other hand, it is probably not to be expected yet either. The following can be concluded from the survey:

- Most of the controlled experiments focus on the inspection and testing activities in isolation, and make theoretical comparisons afterwards. The simulation studies, on the other hand, have to include both activities to able to show the appropriate performance of the modelled process.
- The case studies do all point towards higher effectiveness for inspections, compared to testing, while the experiments not show any evidence that a specific technique is superior to the other.
- Most, though not all, of the different studies conclude that inspections and testing are complementary, even though the classifications of faults are not the same. Both of the research strategies case studies and experiments come to the conclusion that inspection and testing detect different types of faults.

It is concluded that both techniques, inspections and testing, are highly dependent on a number of factors, i.e. several parameters are of interest, when comparing inspections and testing, which make it difficult to draw any general conclusions from the examined studies. A single statement about inspections and testing is not given very easily.

Since the studies often focus on the inspection and testing activities in isolation, the theoretical comparisons are made afterwards. These evaluations often lead to suggestions of applying the activities in combination, rather than isolation. However, few studies do evaluate in detail the activities in combination. How to combine the activities, and how it will affect the product quality are rather unclear. More knowledge has to be gained and the understanding of the relationships between inspections and testing has to increase.

Hence, this knowledge has to be built, and an appropriate way is empirical studies. Studies guided by all methodological strategies are important, since they all are valuable in building a solid empirical foundation.

In the next section, in order to investigate the listed research questions, general empirical research methodology is discussed and the specific techniques and means used in this thesis are described.

3. Research Methodology

Empirical research in the area of software engineering has evolved over the last decades. 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 measurable observations rather than belief.

Even though the field has evolved, the empirical software engineering research has been criticized for still being immature [36]. However, recent guidelines for evaluating situations, methods and techniques are proposed on from several directions [21], [42]. The research methodology is based on the same principles that can be used in other areas, like social, medical, and psychological research [32], but when criticized it is compared to these more mature research fields. The software engineering field is relatively young when compared to these research fields.

When considering research methodology in general, the first step in a successful research study is often to state the research goal and research questions. Thereafter, the research approach is chosen and an appropriate study design decided upon. The chosen design will include the choice of data collection methods and analysis methods. This section describes common methodological approaches for research. Furthermore, the data collection and analysis methods used in the studies in this thesis are described.

3.1 Methodological Approach

Two approaches are mentioned when discussing research strategy, *fixed* and *flexible* research designs [32]. 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, which also is referred as qualitative 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* [32]. 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, possibly all of them.

Dependent on the purpose of the research an appropriate investigation type is chosen. Three major alternatives are recognized [20], [42], *surveys, case studies,* and *experiments* [15].

Surveys: A survey is often referred to as a fixed design research strategy, though can also be of flexible design. 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 in other 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. The results are analysed to be generalized to the sampled population. Hence, the results from an investigation performed in one organization are difficult to generalize to other organizations.

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. Case study as a strategy is defined as using several methods for data collection, where qualitative data most invariably are collected, though; also quantitative data can be included. 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 can 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 cannot be generalized to every situation.

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. Though, for obtaining results that are broadly applicable across many types of projects and processes, the context of the experiment is of high importance.

Dependent on the chosen investigation type for the empirical study, feasible methods for data collection and analysis are decided upon. The following describes the methods used for the research in this thesis.

3.2 Data Collection and Analysis

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. Most often in a flexible design 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. When not focusing on confirming predefined solutions and initial interpretations the overlapping may give new insights and alternative explanations [29]. On the other hand, 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.

The analysis of quantitative data can range from being simply organized to being exposed for some complex statistical analysis. However, qualitative data should also be systematically analysed. According to Robson [32], there does not exist one single accepted convention for qualitative analysis, in contrast to the existence of established statistical methods for quantitative analysis.

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. What has been observed is 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 observation method used in this thesis is mainly of the type participatory observation [6], [32], where the observer participates in the group or situation under study. In papers 2 and 3 the researcher(s) have participated in the daily work in the organization under study. Most important with participatory observations 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.

Interviews: One advantage with data collection through interviews is the flexibility. The researcher has the possibility to follow up 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, like the *fully-structured interview,* over the *semi-structured interview,* to the other extreme, the *unstructured interview* [32]. 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 includes also open-response questions. The semi-structured interview has predetermined questions, but the interviewer can change the order, as well as the wording of the question, and explanations can be given. The unstructured interview has often a topic, which the interviewer poses open questions about.

In this thesis, the performed interviews are of semi-structured and unstructured types (papers 1, 2 and 3).

Content Analysis: A third method for data collection is review of written documents. The content analysis in qualitative inquiry examines relevant records and documents, with the focus to gather information and generate findings that are useful. 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 papers 2 and 3, data have mainly been collected from existing documents. However, content analysis also includes analysing 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 papers 1 and 4.

Simulations: Simulations can be used both as a data collection method [32] 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 [23]. 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 designed by differential equations. The discrete-event paradigm and system dynamics may also be combined as a hybrid model [26]. In combination, the discrete event paradigm shows specific process tasks and describes unique process artefacts, while the system dynamics paradigm is better for representing the project environment and the feedback loops that may exist in the project environment.

System dynamics modelling and simulations have been used in paper 2 and 3 for increasing the understanding of the investigated processes. The simulation in paper 2 also evaluated the usefulness of system dynamics simulations, while paper 3 focuses on the understanding of the software test process.

3.3 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 in a sufficient level of details, 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. The validity of a research study can be evaluated from four perspectives [40].

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.

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 in conjunction with the main contributions in Section 4 and separately in each included paper.

4. Research Results

This section presents the main contributions of this thesis, related to each research question. In addition, the main threats to validity that have been addressed during the studies are discussed.

4.1 Main Contributions

The following reports on the main contributions of this thesis, related to each research question.

RQ1. How do software organizations perform verification and validation?

A general view of how organizations perform their verification and validation cannot be expressed. The procedures of different organizations are most often too heterogeneous. Instead, the research aims at creating a body of knowledge of how different organizations control and manage

their activities, and investigates the organizational attributes that affect the chosen procedures.

Paper 1 presents a qualitative survey investigating the state of practice in 11 Swedish software development organizations. The survey was conducted to increase the understanding of the state of the test process practices in software industry. The data were collected by interviews in software development departments at the participating organizations and thereafter assessed and analysed. Paper 1 is an extension of a previous published paper "Verification and Validation in Industry – A Qualitative Survey on the State of Practice", by Carina Andersson and Per Runeson, [2]. This paper is referred to if more specific information on the practices used in the surveyed organizations is desired. Furthermore, the case studies in paper 2 and 3 have given a more extensive view of current practices in these two investigated organizations.

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 in the smaller organizations. These organizations relied more on experienced people among the employees than on documentation. A threshold was observed somewhere between 30 and 50 developers in the organization. Organizations larger than this breakpoint needed the process to guide and support the work, while in organizations below, less formal means was sufficient.

The development among the surveyed organizations was either incremental (internal release cycles of months) or daily build (internal release cycles of days). Increments were used among more process-focused organizations and daily build was more frequently utilized in less process-focused organizations. Using incremental development or daily build provides an opportunity to the organizations to start testing early during the first developed increments with less functionality. Thus, the cycle time for a release will be reduced since it allows testing in parallel with development.

No specific approach for improvements, related to the used process, could be identified among the surveyed organizations. The approach taken depended on the persons involved, their background and experiences. Test automation was regarded as an improvement area by several of the organizations. Handling the legacy parts of the product and related documentation presented a common problem in improvement efforts for product evolution. The test automation was performed using scripts for products with focus on functionality and recorded data for

products with focus on non-functional properties. However, a key issue regarding test automation is the tested product's stability. There will only be potential savings if the product is long-lasting or the automation scripts are possible to use for a product line.

RQ2. Can the understanding of the verification and validation process, and its activities, be increased by simulations?

The importance of understanding and specifically communicating the verification and validation process and its activities are emphasized by several organizations during the conducted survey (paper 1). Process understanding and improvements are considered to be essential in software industry in order to achieve cost effectiveness and short delivery times.

In this thesis the use of simulation as a tool for increasing the understanding of the process activities is investigated. Specifically, the simulation paradigm system dynamics has been evaluated with respect to its usefulness as a tool for visualizing different factors' impact on the process activities. The contributions to RQ2 are mainly based on the research conducted in the studies reported in papers 2 and 3.

In paper 2, a simulation model was developed to evaluate its usefulness in an industrial setting, and build a foundation for future simulation modelling with the system dynamics paradigm. The model can be used for relocating resources between different process phases, specifically the requirements phase and a merged testing phase. The simulation results show that moving more resources to the early phases of the modelled project should have given reduced cycle time.

Paper 3 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 was adapted and extended to fit an organization. The simulation model was applied for investigating 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 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 of the participating project members is increased due to their involvement.

The studies show advantages with process simulation using system dynamics. As a tool the simulation models performed well by visualizing feedback loops, which are difficult to understand without assistance. Linear relations without feedback are often understandable and a line of thought easy to follow. However, when the relations affect factors earlier in the course of events and creating subsequent loops, a tool for visualizing this behaviour is needed.

RQ3. How do inspections compare to testing regarding performance in finding design faults?

As discussed in Section 1, a combination of different verification and validation activities is means for achieving a high quality software product, developed with low fault injection and exposed to effective fault detection techniques.

The focus of this research question is on the combination of inspections and testing as fault detection activities. The contribution to RQ3 is mainly obtained from the study reported in paper 4, 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. Related work, experiments combining the methods, has focused on fault detection on code artefacts, while the work in this thesis emphasizes the importance of also investigating and comparing the activities on a higher abstraction level.

In the study investigating inspection and testing, the efficiency and effectiveness of the techniques for detection of design faults were evaluated. The general results from this study show that the values for efficiency and effectiveness are higher for the inspection technique and that the testing technique tends to require 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 was 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. The study shows that some faults could be found

by either one of the techniques, though this was not statistically significant.

4.2 Threats to Validity

The contributions discussed in the previous section rely on the conclusions drawn from the results of the studies, which are reported in the included papers. Below, the main validity threats to these conclusions related to each research question are discussed. Furthermore, the threats to validity are discussed separately in each paper, together with a list of strategies to reduce the threats.

General threats to the external validity of a survey of the type in paper 1 concern whether the sample of the study represent an appropriate population. The sample chosen has diversity in several aspects, although it is still a result from convenience sampling in that the organizations were geographically located in southern Sweden. Threats to internal validity might also affect the outcome of the survey. One threat concerns the respondents. They may give different views of the reality depending on their role in the surveyed organizations. In the reported survey, most organizations had more than one respondent. The respondents ranged from the roles of test managers and testers to project managers. As a qualitative study there is potential for bias, from the researchers as well as the respondents. This threat were countered by triangulation [32], i.e. having multiple sources for the data, peer debriefing (having a reviewer acting as a quality assurance person), and member checking (having the material received from the respondents returned to them for review).

In the two simulation studies in papers 2 and 3, the major threat is concerned with construct validity, i.e. whether the models represent the modelled processes or not. The measure taken in these situations was to have an iterative development of the models. As the models were projects specific, representatives from the organizations of the modelled processes validated the models in each iteration.

General threats in experimental studies, like the experiment that evaluates two fault detection techniques in paper 4, often concern external validity, i.e. whether the results are generalizable to other settings. These might be reduced by choosing an appropriate design, and consider the experimental environment and its subjects and objects. In this specific study, a major threat concerned the construct validity. The difficult point was in having the same faults to exist in two different artefacts, and that

these faults still were representative for both artefacts. The faults used in the study origin from the design document, and all were analysed separately to ensure that they could propagate into the code without detection.

5. Future Work

In all of the included papers, possible future research directions are mentioned and in some detail discussed. There are methods and techniques that can be improved and several of the suggested directions may be further investigated. These issues are listed in each paper, and in this section a more focused plan with accompanying research strategies are presented.

Hence, the planned further research takes its starting point in the previous work reported in this thesis, with focus on further evaluating simulations as a method in software engineering research. The specific research goal is to investigate simulation models as decision support in the verification and validation planning. Though, the goal is also to apply the simulation models as a tool to evaluate the benefits, and the trade-off from different choices made during the planning. These goals will be fulfilled by the means of empirical studies combined with the development, building and usage of simulations models.

A plan for further research is presented below, together with purpose and issues to reflect upon. The research plan is summarized in Table 2.

Table 2. Research plan

Research step	Description	Research approach
1, 2	Development and validation of simulation model in an educational environment.	Data collection: content analysis Validation: sensitivity analysis, literature survey
3	Evaluation of the simulation model.	Data collection: interviews, questionnaires, controlled experiment
4	Extension of simulation model in industrial environment, integrated with reliability models.	Data collection: content analysis, interviews Evaluation: interviews

1. *Development of simulation model of a software development process*
   Based on a project in an educational environment, in a forthcoming case study an extension of the template model from paper 3 will be developed. Data will be collected by the means of content analysis from previous conducted projects. The purpose of the study is to investigate the factors that influence and have a major impact on the final software product and the development process.

   In the study, it will be investigated whether a process model, visualizing the different process phases, is the most appropriate way for modelling the software development; or if a product model, visualizing the different artefacts (documents and code) and their evolution is better. Advantages and disadvantages with each approach will be reflected upon, with the aim of understanding the range of situations where each is appropriate and convenient to apply.

2. *Validation of the simulation model*
   The simulation model, mentioned in step 1, will be validated with data from previous projects. A substantial amount of data is available from previously conducted projects in the same educational environment, which makes it possible to conduct an extensive analysis and validation of the model.

   A literature survey of related work in the area will be conducted to analyse which data and information are required for the model validation.

3. *Evaluation of the usefulness of the simulation model*
   It is of interest to investigate to what extent a simulation model can assist project participants as the proposed decision support. Therefore, the model will be qualitatively evaluated regarding usefulness as a planning tool for the project managers. Data collection will be conducted with questionnaires and interviews, to obtain subjective opinions from project participants, compared to the outcome after the projects are finished.

   An experiment will be conducted as a complement to the qualitative evaluation of the model. The experiment design will give an opportunity to keep some variables controlled. Three treatment groups will be arranged: one having access to the model for

planning, one group having access to average data from earlier years' projects, and one final control group without any assistance in the planning phase.

Issues to further reflect upon are the possibilities an experiment gives. It is possible to manipulate other factors of interest, e.g. factors having impact on the status of the product at delivery from developers to testers.

4. *Development of simulation model in an industrial setting*
Based on the evaluation of the previous model, this step will include development of an extension of the model, to suit an industrial setting. The extension will be based on data collected through content analysis from an organization specific fault reporting system and interviews with representatives from this organization.

Furthermore, the extension will be integrated with reliability models. By integrating reliability estimates with the simulation model, a tool for supporting the decisions regarding reliability strategies is obtained. The tool will visualize the impacts of choices on reliability, cost, and schedule. Hence, the purpose is to evaluate the trade-off between reliability on the one hand, and cost and time on the other.

In the following chapters, the papers in this thesis are presented.

6. References

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

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

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

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

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

[8] Brooks, F. P., *The Mythical Man-Month* (Anniversary ed.), Addison-Wesley Publishing Company, 1995.
[9] Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development", *IBM Systems Journal*, 15(3):182-211, 1976.
[10] Genuchten, van, M., "Why is Software Late? An Empirical Study of Reasons For Delay in Software Development", *IEEE Transaction on Software Engineering*, 17(6):582-590, 1991.
[11] Glass, R. L., "The Relationship Between Theory and Practice in Software Engineering", *Communications of the ACM*, 39(11):11-13, 1996.
[12] Hetzel W. C., "An Experimental Analysis of Program Verification Problem Solving Capabilities as they Relate to Programmer Efficiency", *Comput. Personnel*, 3(3):10-15, 1972.
[13] Hetzel, B., *The Complete Guide to Software Testing*, John Wiley & Sons, 1988.
[14] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std. 610.12-1990, IEEE Computer Society, 1990.
[15] Juristo, N., Moreno, A. M., *Basics of Software Engineering Experimentation*, Kluwer Academic Publisher, 2001.
[16] Juristo N., Moreno A. M., 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.
[17] Kamsties, E., Lott, C. M., "An Empirical Evaluation of Three Defect-Detection Techniques", *Proceedings of the 5th European Software Engineering Conference*, pp. 362-383, 1995.
[18] Kan, S. H., *Metrics and Models in Software Quality Engineering*, Addison-Wesley Publishing Company, 1995.
[19] Kit. E., *Software Testing in the Real World: Improving the Process*, Addison-Wesley, 1995.
[20] Kitchenham, B. A., Pickard, L. M., Pfleeger, S. L., "Case Studies for Method and Tool Evaluation", *IEEE Software*, 12(4):52-62, 1995.
[21] Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El Emam, K., Rosenberg, J., "Preliminary Guidelines for Empirical Research in Software Engineering", *IEEE Transactions on Software Engineering*, 28(8):721-734, 2002.
[22] 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.
[23] Law, A. M., Kelton, W. D., *Simulation Modeling and Analysis* (3rd ed.), McGraw-Hill, 2000.
[24] Linger, R. C., Mills, H. D., Witt, B. I., *Structured Programming – Theory and Practice*, Addison-Wesley, 1979.
[25] Madachy, R. J., "System Dynamics Modelling of an Inspection-Based Process", *Proceedings of the 18th International Conference on Software Engineering*, pp. 376-386, 1996.
[26] Martin R., Raffo, D., "Application of a Hybrid Process Simulation Model to a Software Development Project", *Journal of Systems and Software*, 59(3):237-246, 2001.

[27] Montgomery, D. C., *Design and Analysis of Experiments* (3rd ed.), John Wiley & Sons, 1991.
[28] Myers, G. J., "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections", *Communications of the ACM*, 21(9):760-768, 1978.
[29] Patton, M. Q., *Qualitative Evaluation and Research Methods* (2nd ed.), Sage Publications, 1990.
[30] Raffo, D. M., Kellner, M. I., "Analyzing Process Improvements Using the Process Tradeoff Analysis Method", *Proceedings of Software Process Modelling and Simulation Workshop*, 2000.
[31] 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.
[32] Robson, C., *Real World Research* (2nd ed.), Blackwell Publisher, 2002.
[33] Roper, M., Wood, M., Miller, J., "An Empirical Evaluation of Defect Detection Techniques", *Information and Software Technology*, 39(11):763-775, 1997.
[34] Royce, W. W., "Managing the Development of Large Software Systems", *Proceedings of Western Electronic Show and Convention*, pp. 1-9, 1970. (Reprinted in *Proc. of the 9th International Conference on Software Engineering*, pp. 328-338, 1987).
[35] Russell, G. W., "Experience with Inspection in Ultralarge-Scale Development", *IEEE Software*, 8(1):25-31, 1991.
[36] Shaw, M., "The coming-of-Age of Software Architecture Research", *Proceedings of the 23rd International Conference on Software Engineering*, pp. 657-664a, 2001.
[37] So, S. S., Cha, S. D., Shimeall, T. J., Kwon, Y. R., "An Empirical Evaluation of Six Methods to Detect Faults in Software", *Software Testing, Verification and Reliability*, 12(3):155-172, 2002.
[38] Sommerville, I., *Software Engineering* (6th ed.), Addison-Wesley Publishing Company, 2001.
[39] Thelin, T., Runeson, P., Wohlin, C., "An Experimental Comparison of Usage-Based and Checklist-Based Reading", *IEEE Transactions on Software Engineering*, 29(8):687-704, 2003.
[40] Trochim, W. M. K., *The Research Methods Knowledge Base* (2nd ed.), Atomic Dog Publisher, 2001.
[41] Whiting, R., "Development in Disarray", *Software Magazine*, 18(12):20, 1998.
[42] Wohlin, C., Runeson, P., Höst, M, Ohlsson, M. C., Regnell, B., Wesslén, A., *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publisher, 2000.

PAPER 1

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.

1

Abstract

In order to understand the state of test processes practices in the software industry, we have conducted a qualitative survey, covering software development departments at 11 companies in Sweden of different size and application domains. The companies develop products in an evolutionary manner, which means, either new versions are released regularly, or new product variants under new names are released. The survey was conducted through workshop and interview sessions, loosely guided by a questionnaire scheme.

The main conclusions of the survey are that the documented development process is emphasized by larger organizations as a key asset, while smaller organizations tend to lean more on experienced people. Further, product evolution is performed primarily as new product variants for embedded systems, and as new versions for packaged software. The development is structured using incremental development or a daily build approach; increments are used among more process-focused organizations and daily build is more frequently utilized in less process-focused organizations. Test automation is performed using scripts for products with focus on functionality and recorded data for products with focus on non-functional properties. Test automation is an issue which most organizations want to improve; handling the legacy parts of the product

and related documentation presents a common problem in improvement efforts for product evolution.

1. Introduction

Verification and validation activities take a substantial share of project budgets. Early rules of thumb devoted 50% of the time schedule to testing [8], and no great breakthroughs seem to have changed this dramatically. An earlier survey, with a focus on lead-time consumption, concludes that there is a significant shift of the main lead-time burden from programming to integration and testing, when distributing systems [7]. Verification and validation (V&V) are the activities performed during a software development project to ensure that the right system is developed (validation) and that the developed system is right (verification) [5]; hence half of the time is spent checking that what is done during the other half is correct. V&V activities primarily include inspection and testing and in this survey, we focus on the testing part.

The high ratio of time and effort spent on verification and validation seems to be particularly true for product evolution, where development efforts from earlier releases of the product can be reused, but all functionality has to be verified and validated in every release. In order to understand the state of practice of the processes in industry, a qualitative survey was launched, covering software development departments at 11 companies in Sweden of different size and application domains, all evolving products continuously, either releasing new versions of existing products, or new product variants. The survey is conducted through workshop and interview sessions, loosely guided by a questionnaire scheme. The departments range from 6 to 200 developers, in domains of communications, image processing and support systems. In this paper, these departments are referred to as "organizations." They are selected based on availability, but we ensured that there is sufficient variation with respect to size, age and application domain, to draw relevant conclusions based on the findings.

The methodology followed in the study is presented in Section 2 and the surveyed organizations are briefly presented in Section 3. The observations and the analysis are reported in Section 4 and finally, the conclusions are summarized in Section 5.

2. Research Questions and Methodology

The purpose of the survey is to investigate the current status of verification and validation processes in software companies developing systems in an evolutionary manner. However, as a qualitative survey is not an objective study, but a view of the world seen from the researchers' viewpoints, the approach to the survey as well as the specific research questions are biased by the researchers. In order to enable critical reviews of the observations and conclusions of the study, the viewpoints of the researchers are reported here, leading to the research questions investigated.

2.1 Researcher Viewpoints

Below, a few statements summarize the values of the researchers which performed this investigation.

- *Process focus.* The documented process is an important means for communication and capturing experiences. The communication may regard tasks to perform and the progress of the development project. Experience capturing may comprise historical time and defect data, which require a process model as a reference. The term process is here broadly defined to include a variety of methods and other developmental support [17], while the documented process refers to company development manuals and similar documents.
- *Balance between process and people.* All knowledge and experience cannot be captured in a documented process. Software engineering is hence heavily dependent on individuals. The process is assumed to be more important for large organizations, while people are more important for smaller organizations.
- *Inspections*. It is assumed, and to some extent empirically shown [2], that inspections provide efficient means for early defect removal. Inspections are also assumed to contribute to information spreading within a project or an organization.
- *Structured V&V.* It is assumed that a structured approach to V&V would help many organizations and improve their efficiency and effectiveness [13].

- *Product evolution.* Products evolve through their life cycles, which most often are longer than originally planned. Products evolve, either as distinct releases of the same product, or as components in new products in a product family fashion. It is assumed that this is all too often an ad hoc process, both with respect to product and process [6].

These viewpoints are further defined in terms of the set of interview questions raised, and might impact unconsciously in the observations and the interpretation of the observations. Hence this open presentation of the view provides the readers with means to arrive at their own interpretation.

2.2 Research Questions and Method

Based on the researcher viewpoints, and the questions pinpointed for this special issue, we have addressed six research questions:

RQ1. How much is the documented process emphasized by the organizations?

RQ2. Are there any relations between the process emphasis and the characteristics of the organizations or their products?

RQ3. Which criteria govern the selection of the process?

RQ4. Which kinds of evolutionary development exist among the organizations?

RQ5. How is the test automation tailored to support evolutionary development?

RQ6. Which criteria guide the improvement of the process?

In order to address these questions, the survey is guided by qualitative research methodology, hence using a flexible design [20]. This is not intended to be in contrast to the need of quantitative research in software engineering [12], [22]. Instead, the methodologies are expected to complement each other. Quantitative methodology is better suited for studies on, for example, specific methods or notations, while qualitative methodology is better suited for broader studies that seek to present overviews or more generalized information. The analysis of research questions 1–3 and 6 is also reported by Andersson and Runeson [1], with

more detailed observations, seen from a more general V&V process viewpoint.

A general model for qualitative studies is shown in Figure 1. The data in this study was (a) collected using unstructured interviews, to some extent in the form of group interviews [20]. Data are recorded by two researchers taking notes from the interviews[2]. Hence some data reduction (b) was conducted during the observations. Further data reduction occurred in a later step based on the topics selected for deeper analysis. The data collection procedures are presented in more detail in Section 2.3.

The observations were compiled into reports which were sent to the interviewees for feed-back, who thus acted as critical reviewers. A third researcher was also a critical reviewer in the analysis (d).

The analysis (c, e) was conducted by using a conceptually clustered matrix [20]. Data was recorded in a matrix in columns related to the issues in the interviews. The data analysis procedures are presented in more detail in Section 2.4.

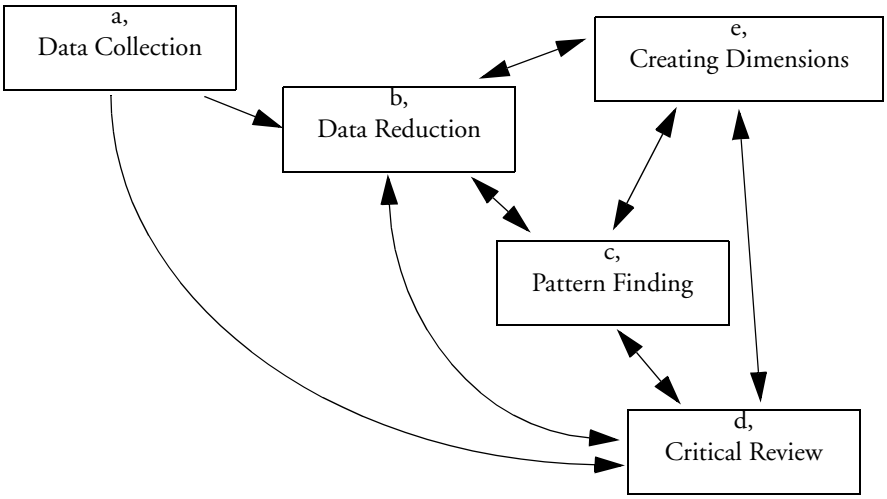


Figure 1. *A general model for qualitative studies [15].*

2. The interviews with organizations *Phone* and *Security* were recorded by only one researcher.

2.3 Data Collection Procedures

The survey was conducted in two cycles. First, five organizations were surveyed using the workshop format. The organizations attended a workshop series, organized within a local software process improvement network (SPIN)[3]. The attendants assigned themselves to the workshop based on their interests. The attendants belonged to quality departments or test departments, and most attendants were present at most workshop meetings.

The workshop cycle was conducted as follows:

1. The workshop host presented the company V&V activities. The hosts were free to present any topic, as long as they included a list of strong and weak issues regarding their V&V process.
2. The researchers checked that a list of questions was covered [1] and raised questions that were not voluntary addressed.
3. The researchers summarized the meeting in a report, which was proof-read by the company representative.
4. The findings were analyzed, compiled into a joint report and fed back to the organizations.

In the second cycle, another six organizations were surveyed using a more direct interview format. Those organizations were approached by the researchers and selected to achieve diversity with respect to size, age of organization, and application domain. This cycle was conducted in a similar manner, except that the only participants at the meeting were the interviewees and the researchers. Finally, within the second cycle, one of the first five organizations was interviewed again (*Read*, see Table 1), since they were at a point of major process change at the time of the first interview. The second interview was conducted six months later, when some of the changes had been implemented.

2.4 Analysis Procedures

Data analysis was conducted in three cycles. First, the workshop data collected was summarized in a rather informal report, for internal use

3. http://www.spin-syd.org

amongst the workshop participants. Second, an analysis of the survey in general was conducted, as reported in [1]. Thirdly, the analysis was conducted towards the specific focus of test processes for evolutionary software development. The analysis was mainly conducted by two researchers, and a third researcher acted as a critical reviewer.

The procedures can be summarized as follows, with references to Figure 1:

1. The data was structured in a matrix with rows for each participating organization and columns representing different aspects covered during the interview (b). The first set of columns were selected from the areas in the questionnaire checklist (e). The data was primarily taken from the compiled report and secondly from the original notes from each interview.

2. The data in each column was then analyzed; categories were sought and ranked with respect to different criteria (e).

3. The analysis was made based on the matrix; relations as well as lack of relations were sought (c).

4. The analysis was cross-checked by a third researcher, acting as a critical reviewer of the procedures and the analysis (d).

2.5 Validity

In empirical studies, the concept of validity is central [20], [22]. The validity analysis seeks to identify threats to the research's validity, and proposes actions to be taken to improve validity. Different models for validity classification exist. Here we adhere to the model originally presented by Lincoln and Guba [16], [20].

The model divides threats into validity in three broad headings, *reactivity, respondent bias* and *researcher bias*. *Reactivity* and *respondent bias* include the risk that the respondent acts differently than normally, for example, acting differently due to the researchers' presence, or answering to fulfill the researchers' expectations, instead of answering truthfully. *Researcher bias* refers to the preconditions and assumptions the researchers bring into the situation, which may affect, for example, data collection or analysis.

To reduce threats to validity, different strategies can be implemented, addressing different kinds of threats. A list of such strategies [20] for dealing with different threats is presented below.

- *Prolonged involvement* means that the researcher follows the respondents for a longer period of time, to get acquainted with the studied environment. This strategy is not explicitly used in the current study, although there is a long cooperation between one of the researchers and the studied organizations in general within the local network, reducing the threats of reactivity and respondent bias while, on the other hand, increasing researcher bias.
- *Triangulation* means having multiple sources for the data. In the current study, *observer triangulation* is used, i.e. two researchers are present at almost all interviews. Furthermore, the interviews include triangular questions: direct questions where the interviewees are asked how they perform their V&V, and indirect questions where they are asked what is good and bad in their V&V process.
- *Peer debriefing and support* refers to having peers cross-check the analysis and act as a coach to the researchers. This strategy was used in this study, having two researchers in the data collection and analysis, and a third reviewer acting as a quality assurance person.
- *Member checking* means returning material to the respondents for feed-back. This is used in the current study by reporting the results back to the subjects of the study in writing and in seminars.
- *Negative case analysis* refers to "playing the devil's advocate." This strategy is applied in the study, requiring the third researcher try to find alternative cases as explanation to the variation observed in the data.
- *Audit trail* means keeping full record of the activities during the study. This is a weakness of the current study, as the interview material is not tape-recorded. However, the researcher view is presented (see Section 2.1), the investigation procedures are openly reported (see Section 2.3 and Section 2.4), data collection is documented, although not publicly available for confidentiality reasons.

The presented observations reflect the survey participants' answers, i.e. it is the organizations' own picture which is presented, with a risk that it is polished. The participants might have given answers, which may not

accurately reflect the situations at the associated organizations. However, this risk is limited in the current study, as the participants had nothing to gain from polishing the truth. Furthermore, within the software process improvement network, where most of the surveyed organizations take part, there is a tradition of openness between peers as well as towards the researchers and other external sources. Hence, we cannot find any reason that the survey participants would polish the truth, nor do we believe there is a difference in this respect between the workshop and the interview sessions.

3. Surveyed Organizations

Our sample in the survey comprised departments at 11 Swedish companies. They represent a diverse selection of application domains, product types and company characteristics, although they are not systematically sampled from any larger distribution. As they sometimes are just very small parts of large companies, we refer to them as organizations. The surveyed organizations are listed in Table 1 in decreasing size order, and referred to by pseudonyms for confidentiality reasons. The terms used to characterize the organizations are defined below.

3.1 Product Characteristics

The products developed by the surveyed organizations are characterized along three dimensions.

The *application domains* of the surveyed organizations are of five kinds.

- Radar image processing means a radar system for surveillance of large areas.
- Communication involves networked products as well as wireless communication products.
- CASE tools refer to software tools for developing other systems, either for general software systems, or for specific instances of systems including hardware components.
- Image processing means products which take pictures and analyze them for different purposes.

Table 1. Participating organizations.

	Product characteristics			Process characteristics				Organizational characteristics			
Pseudonym	**Application domain**	**Product type**	**Product value**	**Process emphasis**	**Process structure**	**Evolution strategy**	**Automation approach**	**Customer type**	**Business model**	**Size**	**Age**
Radar	Radar image processing	Embedded	Non-functional	++	Increment	Variants	Recorded data	Defense	Contract	200	>20
Phone	Communication	Embedded	Functional	+	Increment	Variants	Scripting	Private	Market	150	~15
Network	Communication	Embedded	Functional	+	Increments[a]	Variants	Scripting	Business	Market	120	18
Automation	CASE tools and control	Packaged	Functional	++	Daily build	Versions	Scripting[b]	Business	Market	80	~25
CASE	CASE tools	Packaged	Functional	0	Increment	Versions	Scripting	Business	Market	50	11
Read	Image processing	Embedded	Non-functional	-	Daily build	Variants	None	Private	Market and contract	30	6
Security	Image processing	Embedded	Non-functional	-	Daily build	Variants	Recorded data	Business	Market	20	5
Monitor	Image processing	Embedded	Non-functional	-	Daily build	Variants	Recorded data	Business	Market	20	3
Product	Support systems	Packaged	Functional	--	Daily build	Versions	Recorded data	Internal	Contract	7	4
Sales	Support systems	Packaged	Functional	--	Daily build	Versions	Scripting	Internal	Contract	7	10
SubSales	Support systems	Packaged	Functional	--	Daily build	Versions	None	Subcontractor	Contract	6	11

a. One option for projects.
b. Trial.

- Support systems are business information systems, intended to support different kinds of business.

The *product type* defines how the software is offered to the market. Embedded software refers to software which is embedded in some sort of equipment, including the hardware which the software runs on. Packaged software is sold as a separate unit and is installed by the user on, for example, a PC or workstation platform.

The *product value* may be functional or non-functional. A PC application has primarily, for example, its value in the functions provided to the user, not that the reliability of the software is very high. On the contrary, in an embedded system, for example for image processing, non-functional properties like the quality of the image or the speed of the image transfer are of primary interest to the user, while the list of functions in such a product is shorter.

3.2 Process Characteristics

The processes used in the surveyed organizations are characterized with respect to four dimensions, which are further analyzed in Section 4.1 through Section 4.4:

The organizations *emphasize* the value of the documented process differently. The organizations are classified in five classes (– –,–, 0, +, ++) from no emphasize of a documented process to very much emphasis on an extensively documented process.

The *process structure* is characterized as either daily build or incremental. In *daily build*, all changes are made available to the whole project on a daily basis; in *incremental development*, new versions comprising a set of implemented functions are delivered to integration and system testing at specified time interval, typically monthly.

The *evolution strategy* is the strategy applied to develop new products, either:

- new *versions* of a product are delivered at various intervals, generally under a new version number (e.g. 1.3, 2.0).
- new *variants* of products are delivered, generally under a new name (e.g. X200, X300).

The test *automation approach* used is either based on recorded data or on scripts. Using recorded data means that *input data* to the product is

not taken from the real environment, but from an earlier execution of the system in its environment. Using scripts means running programs that execute the system under test, feeding *events* or *input actions* to the system.

3.3 Organizational Characteristics

The surveyed organizations are characterized according to four different aspects.

Customers are either defense, business, private or internal. This variation implies that different kinds of time constraints, market requirements and business models are represented.

- Defense customers comprise large, long-term contracts between customer and supplier.
- Business customers buy products on contract or off-the-shelf for business use.
- Private customers buy products for private use off-the-shelf.
- Internal customers use products developed by another department of the company.

Business models are either market, i.e. the products are offered to a wide variety of customers, or contract, i.e. the product is developed and sold to a specific customer.

The *sizes* of the organizations include all engineers involved in the development, including test staff. The companies as such may be much larger, but here only staff related to the product development departments are counted.

The *age* of the organizations reflect the number of years the surveyed activities in each company have been active.

4. Observations

Observations and subsequent analyses regarding the test processes are reported in this section. The observations are presented, structured according to the research questions in Section 2.2.

4.1 Process Emphasis

During the workshop and interview sessions, discussions concerning the test process proceeded from a general test process, according to Figure 2, since all of the organizations had some model to follow, though not always documented. All organizations were able to map their activities into this general model, though with different degrees of formality. The survey displayed a spectrum of process definitions, ranging from a very well defined process at *Radar* to a very informal and unemphasized definition at *SubSales*. Brief characterizations of each organization's process are presented in [1].

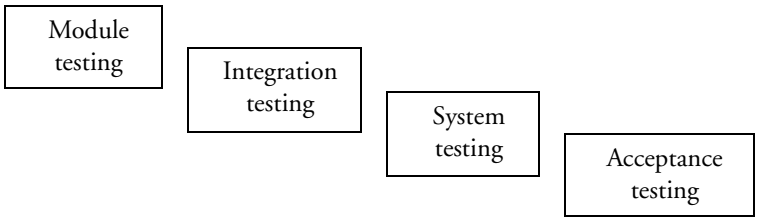


Figure 2. *The general test process.*

One clear observation from the survey is that a documented process is considered an important asset among the large organizations, for testing and for development in general, while smaller organizations rely more on experienced individuals. Table 2 shows the relationships between organization size and the degree of emphasis of the process.

This might be an obvious observation. However, some thresholds are also observed with respect to size and process emphasis. Organizations smaller than 10 developers have almost no process structure at all. They rely on skilled and experienced people. Organizations of size 20-30 begin to identify the need for more structure. They start defining templates and some basic procedures. Organizations of size 50-100 rely on rather well defined templates and common process step definitions. Organizations larger that 150 stress the importance and value of well defined work instructions.

The two organizations that emphasize the process the most are both somewhat special. *Automation* is a part of a larger body and has processes in common across the corporation. *Radar* has through its operations in

Table 2. Process emphasis versus number of developers.

		Number of developers			
Process emphasis		**1-10**	**20-30**	**50-100**	**150-200**
Emphasized	++			*Automation*	*Radar*
					Phone
	+				*Network*
	0			*CASE*	
			Security		
	–		*Monitor*		
			Read		
		Product			
Not Emphasized	– –	*Sales*			
		SubSales			

the defense domain for a long time maintained high requirements for well documented processes.

A common opinion among the organizations is that it is important to increase the visibility of the verification and validation process and allow the members of the organizations to understand the importance of verification and validation. Communicating the process is of high priority and often a part of the ongoing improvement work.

In summary, there is a wide range of different attitudes towards the value of the process (RQ1), and the most visible relation to the process emphasis is the size of the organization (RQ2). There seems to be a breakpoint somewhere between 30 and 50 developers in the organization, over which the process is needed to guide and support the work, while below the threshold, the inter-project communication is sufficient using less formal means.

4.2 Process Selection

Some organizations have chosen an explicitly incremental development process model [21]. Subsets of functionality are defined as increments to allow better control over the projects and to reduce risks. *Radar, Phone* and *CASE* have such models (see Table 3). In *Network,* the projects may choose an incremental model as one of the options. Other organizations have smaller, less defined increments, more towards the daily build

Table 3. Process emphasis versus process structure.

	Daily build	**Increment**
Emphasized [0, +, ++]	*Automation*	*Radar* *Phone* *CASE* *Network*[a]
Not Emphasized [–, – –]	*Security* *Read* *Monitor* *Product* *Sales* *SubSales*	

a. One option for projects.

principle [10], [18]. This kind of approach is used by *Automation, Read, Security, Monitor*, and *SubSales*. *Read* and *Phone* have made pilot use of the extreme programming concept [3] in single projects. Organizations with more emphasized processes tend to use the incremental approach while the less emphasized processes are of a more daily build character (see Table 3). This relation is quite natural as incremental development requires a more well-structured approach, while daily build is the bottom-up solution to avoid big-bang integration problems. The exception, *Automation*, has a tradition of using daily build for approximately 20 years; this policy was established when the company still was young and small.

The incremental and daily build approaches both introduce evolutionary product development into the internal development process as such. The principles are the same, although the time scale is different (see Table 4).

Common to the approaches are that new versions are delivered to a stakeholder at regular basis; in *daily build*, all changes are made available to the whole project on a daily basis; in *incremental development*, new versions comprising a set of implemented functions are delivered to integration and system testing at specified time interval, typically monthly. *Product evolution* implies new releases are delivered to the market, typically once or twice a year. Using daily build or incremental development provides the opportunity to start testing early on the earlier increments with limited functionality, and thus reduce the cycle time for a release since it allows testing in parallel with development.

Table 4. Overview of evolutionary approaches.

Approach	Description	Time scale
Daily build	New versions of the systems are delivered internally	Day(s)
Incremental development	New versions are delivered to integration	Months(s)
Product evolution	New versions of the system are delivered to the market	0,5-1 Year

Network has tried the incremental approach and found that it was not profitable. The hand-over from development to testing became fuzzy, and software which was not fully unit tested was delivered to system test. Hence, they had to identify multiple failures which otherwise were removed by the developers. The other organization using incremental development did not experience such problems.

In summary, incremental development is used by organizations with an emphasized process, while the daily build approach is primarily used by organizations with less emphasized process (RQ3).

4.3 Product Evolution

All of the surveyed organizations develop products that evolve over time. We have identified two distinct approaches to product evolution. Either they release new *versions* of a product, or they deliver new *variants* of products, see Table 6. Versions are marketed under the same product name, but with a new extension (1.3, 2.0), while variants are marketed as new products under a new name (X100, X300). The new versions and variants must differ substantially from the earlier ones, in particular for organizations with private or business customers as their market. The motivation for buying a new product with embedded software, or upgrading to a new release of packaged software must lie in added value for the user, in terms of better non-functional properties or new functionality. Upgrades are offered to the market, however, not bought by every customer and thus multiple versions of the product have to be supported.

In addition to the major releases, new versions of the software with minor changes (polishing), can be developed. This is referred to as "product care" by one of the interviewees. In the case of packaged software, "patches" or "service packs" can be distributed to customers, while in the case of embedded software, new versions of the software are

introduced in the production process gradually. Already delivered products are updated only when very critical faults are detected.

In both the version and the variant cases, a large portion of the software remains the same, and already invested test effort could be saved, either by considering software components well tested "by use" or by automating regression tests. The former approach involves a risk, clearly illustrated by the Ariane 5 failure [11]. The latter approach is used by the surveyed organizations to different extents, but in no case to a very large extent.

It is concluded from the survey, that embedded products are mainly developed using the *variant* strategy, while packaged software is mainly developed using the *version* strategy, see Table 5. The first case is quite natural, as long as downloading new software dynamically in embedded products is not permitted to the user. The second case is not bound by these kinds of technical constraints. However, there are indications that changes are under way. In the *Phone* case, a new version of the software is offered to the users of a specific model of an embedded product, and new variants are planned to allow dynamic download of e.g. Java programs. This trend is indicated by an arrow in Table 5. Further, the *Sales/SubSales* product is built on a general software framework, which is intended to be used as a basis for other variants of products, i.e. approaching the *variant* approach for a packaged software product. This trend is indicated by the second arrow in Table 5. More details on the different evolution strategies are presented in Table 6.

Table 5. Product type versus evolution strategy.

	Variant	**Version**
Embedded	*Radar*	
	Read	
	Security	
	Monitor	
	Phone	–>
	Network	
Packaged		*Automation*
		Product
		Sales
	<–	*SubSales*
		CASE

Table 6. Brief overview of evolution strategy for each company.

Radar	*Radar* develops three variants of their product for three different customers. From a software perspective, they are not managed as a product line [6]. Instead, the development chain was branched for each variant and is not merged again. However, information about identified faults in one variant are fed back to the development of the other variants.
Phone	*Phone* uses the variant approach to develop new products. The same hardware and software architecture is used in several variants of the product.
Network	*Network* develops products using a planned product line approach [6]. They develop products whose appearance and usage are quite different, while they internally are built on the same hardware and software platform.
Automation CASE	*Automation* and *CASE* both release new versions of their product regularly. As the customers have to pay for the new release, or foresee problems with the introduction of new releases, not all customers upgrade for every release, implying that many subsequent releases have to be supported in parallel.
Read Security Monitor	*Read, Monitor* and *Security* have the same origin. The products of the three organizations share processor and optical devices for image processing, while the analysis algorithms and the functionality of the products are very different. The common parts are rather stable and not further developed very much any more, and are primarily considered a platform.
Product Sales SubSales	*Product* and *Sales* (and hence *SubSales*) deliver new releases of a software package at pre-determined dates. The customer is internal or semi-internal, thus avoiding the requirements of real market customers. In the *Product* case, the delivery date is negotiable if the development runs out of time. In the *Sales* case, features are negotiable but not delivery dates, as the system supports the sales department, and one frequent reason for a new release is that a new price list is introduced from a certain date.

In summary, there are two kinds of evolutionary strategies, variants and versions (RQ4), where the former is primarily used for embedded products and the latter for packaged software, although the borderline is under change. In addition, effort is spent on "product care."

4.4 Test Automation in Product Evolution

Test automation is mentioned by most of the surveyed organizations as an improvement area. Most organizations automate testing to some extent, but they are not satisfied with the level of automation.

The automation approach is different, depending on the product characteristics presented in Section 3.1. We concluded from the data that there is a tendency towards believing that non-functional properties constitute the key value of the embedded products, and that the

functionality constitute the key value of the packaged software (see Table 7). The two exceptions, *Phone* and *Network*, where functionality constitute the value for embedded products, are mature communication products. In these domains, the characteristics are taken for granted by the users, and the functionality constitute the competitive advantage for new products. Hence the products are on the borderline of having their focus on the functionality from the user point of view, even though the developers still consider the non-functional properties to be of great value.

Table 7. Product type versus product value.

	Embedded	**Packaged**
Non-functional	*Radar* *Read* *Security* *Monitor*	
Functional	*Phone* *Network*	*Automation* *Product* *Sales* *SubSales* *CASE*

Two different approaches to test automation exist among the surveyed organizations. Either the execution of features is automated by *scripts* running the application, or the inputs to the systems exist as *recorded data* from a real environment. Using scripts means running programs that execute the system under test, feeding *events* or input *actions* to the system. For example, the communication products are tested by running a script which sends commands to the program, which in the operational environment comes from the user. Using recorded data means that input *data* to the product is not taken from the real environment, but from an earlier execution of the system in its environment. For example, the image processing products are tested using previously recorded images to analyze.

Product value is connected to the type of automation approach chosen. Table 8 shows an overview of approaches chosen by the organizations. Products with functional focus are tested using scripts and products with non-functional properties in focus are tested using recorded data.

Table 8. Product value vs. automation approach.

	Recorded data	Scripting	None
Non-functional	*Radar* *Security* *Monitor*		*Read*
Functional	*Product*	*Phone* *Network* *Automation*[a] *CASE* *Sales*	*SubSales*

a. Trial

Organizations that do not use an automation approach have, in most cases, a good reason for not doing so.

- The *Read* product involves a feed-back loop, changing the parameters for the data recording based on the analysis of the data, e.g. to compensate for different light conditions. Hence, earlier recorded data are not realistic enough to use as test data.
- *SubSales* is primarily responsible for module tests, while *Sales* is responsible for the integration and system tests. There is a planned improvement effort to run the automated tests also at the module level.

Although most organizations already perform some form of automated tests, this issue is among the mostly mentioned improvement area. A key issue regarding automation, mentioned by *Sales* is product stability. Investments in test scripts are large and are only worthwhile for a long-lasting product or product-line.

In summary, there are two kinds of test automation among the surveyed organizations, recorded data for products with focus on non-functional properties and scripting for products with focus on functionality (RQ5).

4.5 Test Process Improvement

Most of the surveyed organizations report an intention to improve the test process, although the approach is not very structured, nor very much

emphasized. Examples of improvement initiatives are presented in Table 9.

The improvement approaches taken are very different. The only rationale for choosing a certain approach that we could observe during the survey, is that the selection depends on the persons involved.

One specific issue observed in product evolution regarding process improvement, is what happens when, for example, the requirements specification is improved for the functionality of a new release, while the requirements for the old parts remain badly specified as it is too costly to

Table 9. Brief overview of improvement initiatives for each company.

Radar	*Radar* works on improving the efficiency of the system testing. Test planning methods based on factorial designs [9], [4] are used and found efficient. Improvement actions are driven by a joint university-industry research project.
Phone	*Phone* has recently turned towards incremental development to reduce risks and allow flexibility in which functionality is delivered in certain products. Testing can be started much earlier and thus identify integration problems sooner.
Network	The *Network* quality department makes process guidelines available to the projects but does not require a specific approach be chosen. This liberal attitude may be related to the fact that the company is not part of a larger corporation, like *Radar* and *Phone* are.
Automation	*Automation* has, based on the outcome of the workshop behind this survey, initiated a project to store test cases in a database, together with some structuring and test time information, to allow more efficient regression testing. The initiative is taken by interested individuals.
CASE	*CASE* has after its recent merger, produced a new and more structured set of test documents. The new set is based on the best practices in the two merged organizations.
Read	Read has during the period of this survey, assessed their test process and designed a new one. The person responsible for process improvement was in a previous employment working on documenting and structuring the process; hence he chose this approach in his new affiliation. However, as the company was restructured during the period, and reduced in size, only some basic parts of the proposed process are introduced.
Security	*Security* has used the Test Process Improvement (TPI) approach [14] to assess their current status and find improvement areas.
Monitor Product Sales SubSales	*Monitor, Product* and *Sales/SubSales* are working on hands-on improvement actions, like document standards etc. The companies are the smallest and have no tradition of working with defined processes or company standards.

update the requirements for the legacy part of the product. Testing based on the requirements can then be improved for the new parts, while the basis for the regression testing of the legacy part remains bad.

In summary, the approach chosen to process improvement is not based on any of the observed characteristics, but seems to be ruled by the involved persons' background and experience (RQ6).

5. Summary and Conclusions

Verification and validation of software systems take a substantial share of project resources as well as lead-time for a project. In product evolution, this is assumed to be particularly true, as the old parts of the products have to be regression tested to verify that they still work as intended with new functionality added. In order to understand how the verification and validation processes are constituted at organizations of different size and different application domains, a qualitative survey was launched. As all of the organizations perform product evolution, this is an aspect of particular focus in the survey.

The first observation from the study is that there is a wide range of attitudes towards the value of the process, and that larger organizations tend to emphasize the process more than smaller organizations do. There seems to be a breakpoint somewhere between 30 and 50 developers in the organization with respect to the emphasis of the documented process (RQ1 and RQ2).

The evolution approach taken among the organizations is either *incremental* (internal release cycles of months) or *daily build* (internal release cycles of day(s)). It is concluded that organizations which emphasize the documented process use incremental development, while organizations which have a less emphasized process use daily build (RQ3). One organization differs from the pattern. This organization has been using daily build for 20 years, and has turned towards more process emphasis later during growth and mergers with other companies.

It is concluded that there are two different kinds of product evolution, for *embedded* software and for *packaged* software (RQ4). New releases of embedded software products constitute a new product in a product family. The products are sold to new customers or to existing ones; in both cases, a new "thing" is delivered to the customer. In the packaged software case, new versions are mostly delivered as upgrades to existing

customers or as new deliveries to new customers, although there are indications of other combinations of the two. As all customers are not willing to upgrade immediately, different versions have to be supported in parallel. In addition to new variants and new versions, *product care* is performed which implies minor changes to an embedded product or patches to packaged software.

Test automation is considered an area where there are potential saving in product evolution, and they are exploited to some extent (RQ5). We have identified two kinds of test automation, *recorded data automation* and *scripting automation*. It is concluded that products which focus on *non-functional properties* are primarily tested using recorded data, while products with a focus on *functionality* are tested using scripts for automation. The two observed exceptions from this rule are both mature products with a focus on non-functional properties, in which the users take the non-functional properties for granted. Hence the products are on the borderline and maintain their focus on the functionality from the user point of view, even though the developers still consider the non-functional properties be of greater value.

We can conclude from the survey that there is no observed relation between the product characteristics, nor the process focus on which approach is taken to improvement. It seems to be very much dependent on the persons involved, their experiences and their personal viewpoints (RQ6).

In summary, there are a set of specific issues regarding product evolution observed among the surveyed organizations, although the potential savings regarding, for example, regression testing are not fully exploited. This paper, and the workshops and interviews behind it, contribute to an increased awareness of the variation factors in product evolution, a first step towards an improved test process for product evolution.

Acknowledgement

The researchers are thankful to the participating companies for their contribution. Thanks also to Daniel Karlström and Thomas Olsson at the Department of Communication Systems at Lund University for reviewing an earlier version of the analysis report and being involved in discussions on the topic. Thanks to the reviewers and editors of the

special issue for valuable comments. This study was partially funded by The Swedish Agency for Innovation Systems (VINNOVA) under grant for The Center for Applied Software Research at Lund University (LUCAS).

6. References

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

[2] Basili, V. R., Selby, R. W., "Comparing the Effectiveness of Software Testing Strategies", *IEEE Transactions on Software Engineering* 13(12):1278–1298, 1987.

[3] Beck, K., "Embracing change with Extreme Programming", *IEEE Computer*, 32(10):70–77, 1999.

[4] Berling, T., Runeson, P., "Application of Factorial Design to Validation of System Performance", *Proceedings of the 7th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pp. 318–326, 2000.

[5] Boehm, B., "Software Engineering: R & D Trends and Defence Needs", *Research Direction in Software Technology* (P. Wegner ed.), Cambridge, MA, MIT Press, Chapter 19, 1979.

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

[7] Bratthall, L., Runeson, P., Adelswärd-Bruck, K., Eriksson, W., "A Survey of Lead-Time Challenges in the Development and Evolution of Distributed Real-Time Systems", *Information and Software Technology*, 42(13):947–958, 2000.

[8] Brooks, F. P., *The Mythical Man-Month*, Addison-Wesley, 1975.

[9] Cohen, D. M., Dalal, S. R., Parelius, J., Patton, G. C., "The Combinatorial Design Approach to Automatic Test Generation", *IEEE Software*, 13(5):83–88, 1996.

[10] Cusumano, M. A., Selby, R. W., *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People,* Free Press, 1995.

[11] European Space Agency, *Ariane 5 flight 105 inquiry board report*, No 33–1996, http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf (last revisited 02-09-26).

[12] Juristo, N., Moreno, A. M., *Basics of Software Engineering Experimentation*, Kluwer Academic Publishers, 2001.

[13] Kit. E., *Software Testing in the Real World: Improving the Process*, Addison-Wesley, 1995.

[14] Koomen, T., Pol, M., *Test Process Improvement, A practical step-by-step guide to structured testing*, Addison-Wesley, 1999.

[15] Lantz, A., *Intervjuteknik* (Interview methods – in Swedish), Studentlitteratur, 1983.

[16] Lincoln, Y. S., Guba, E. G., *Naturalisitic Enquiry*, Newbury Park and London, 1985.

[17] Lindvall, M., Rus, I., "Process Diversity in Software Development", *IEEE Software*, 17(4):14–19, 2000.
[18] McConnell, S., *Rapid Development: Taming Wild Software Schedules,* Microsoft Press, 1996.
[19] Porter, A. A., Siy, H. P., Toman, C. A., Votta, L. G., "An Experiment to Assess the Cost-Benefits of Code Inspections in Large-Scale Software Development", *IEEE Transactions on Software Engineering* 23(6):329–346, 1997.
[20] Robson, C., *Real World Research* (2nd ed.), Blackwell, 2002.
[21] Sommerville, I., *Software Engineering*, Addison-Wesley, 2001.
[22] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., Wesslén, A., *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, 2000.

PAPER 2

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.

2

Abstract

This paper presents a study with the intent to examine the opportunities provided by creating and using simulation models of software development processes. A model of one software development project was created through means of system dynamics, with data collected from documents, interviews and observations. The model was simulated in a commercial simulation tool. The simulation runs indicate that increasing the effort spent on the requirements phase, to a certain extent, will decrease the lead-time and increase the quality in similar projects. The simulation model visualizes relations in the software process, and can be used by project managers when planning future projects. The study indicates that this type of simulation is a feasible way of modelling the process dynamically although the study calls for further investigations as to how project or process managers can benefit the most from using system dynamics simulations.

1. Introduction

This study was performed in cooperation with Ericsson Mobile Communications AB and is based on a development project carried out in 1999.

As a step in the constantly ongoing work with quality improvements at Ericsson this study was made to show if simulation can be used for visualizing how different factors affect the lead-time and product quality, i.e. number of faults. One of the most important factors that affect the lead-time of the projects and the product quality is the allocation of human resources to the different process phases. Thus, the focus of this simulation study is on resource allocation.

Simulation is commonly used in many research fields, such as engineering, social science and economics. That is, simulation is a general research methodology that may be applied in many different areas. Software process modelling and improvement is, of course, no exception and simulation has started to gain interest also in this area. For example, in [4] a high-maturity organization is simulated with system dynamics models, and in [6] a requirements management process is simulated with a discrete event simulation model. In [8] an overview of simulation approaches is given.

There are several advantages of building and simulating models of software processes. By simulation new knowledge can be gained that can help to improve current processes. Simulation can also be used for training and to enforce motivation for changes.

The objectives of the study that is presented here are to investigate if it is possible to develop a simulation model that can be used to visualize the behaviour of selected parts of a software process, and to evaluate the usefulness of this type of models in this area. The objective of the model is to identify relationships and mechanisms within a project. The study is focused on the tendencies of the simulation results and not the quantitative aspects.

The outline of the paper is as follows: In Section 2 the method used in this study is described. Section 3 describes the execution of the simulation study. Section 4 presents the results of the simulation and Section 5 discusses and summarizes the results of the study.

2. Method

This project was designed as a case study. Case studies are most suitable when data is collected for a specific purpose and when a subgoal of the study is establishing relationships between different attributes. A main activity in case studies is observational efforts.

With support from existing results in literature [3], [16], the research approach was created in three consecutive steps: problem definition, simulation planning and simulation operation. This methodology is based on the process chain concept, but due to lack of enough available, reliable data, the process in practice went into an interactive pattern.

In the first phase, problem definition, the problem was mapped. Then through deeper definition and delimitation, an agreement was created around the study's purpose.

The main part in the second phase of the study, simulation planning, was to identify factors influencing the product quality. The assigner of this study wished to test the idea of using simulation models and this was governing in the details of the study. This was natural as most of the ideas to the quality factors were picked up from the organization's project, through interviewing the project staff and through documents. To add a broader perspective, results and ideas were taken from software literature. Influence diagrams were built including the different quality factors' relation to each other, but primary their effects on lead-time and product quality.

The third phase, operating the simulation model, started with translating a small part of the theoretical model into the simulation tool. A short test showed that the simulation tool worked properly. More features were added from the theoretical model into the simulation tool and more test runs were performed. The verification and validation of the model was made stepwise through the input of the whole model into the simulation tool, and the yardstick to compare with was given by documents and discussions with the assigner.

3. Developing the Simulation Model

In the simulation domain there are two main strategies: continuous and discrete modelling. The continuous simulation technique is based on system dynamics [1], and is mostly used to model the project

environment. This is useful when controlling systems containing dynamic variables that change over time.

The continuous model represents the interactions between key project factors as a set of differential equations, where time is increased step by step. In the standard system dynamics tools, these interconnected differential equations are built up graphically. A system of interconnected tanks filled with fluid is used as a metaphor. Between these tanks or levels there are pipes or flows through which the variables under study are transported. The flows are limited by valves that can be controlled by virtually any other variable in the model. Both this mechanism and the level-and-flow mechanism can be used to create feedback loops. This layout makes it possible to study continuous changes in process variables such as productivity and quality over the course of one or several projects. It is however more problematic to model discrete events such as deadlines and milestones within a project [9], [10].

In the discrete model, time advances when a discrete event occurs. Discrete event modelling is for example preferred when modelling queuing networks. In its simplest form, one queue receives time-stamped events. The event with the lowest time-stamp is selected for execution, and that time-stamp indicates the current system time. When an event occurs an associated action will take place, which most often will involve placing a new event in the queue. Since time always is advanced to the next event, it is difficult to integrate continually changing variables. This might result in instability in any continuous feedback loops [9], [10].

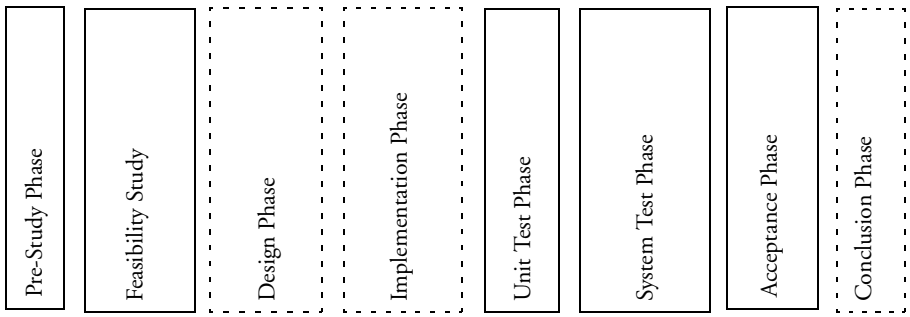


Figure 1. *Process description*

To suit the purpose of this study, which is to visualize process mechanisms, continuous modelling was used. The continuous model was chosen in order to include systems thinking [13] and because it is better suited than the discrete event model at showing qualitative relationships.

3.1 Problem Definition

The study is based on a process that is similar to the waterfall model [14]. The whole process is shown in Figure 1, but the simulation model was focused on the requirements phase and the test phase. The other phases, with dotted lines in Figure 1, were excluded to get a less complex model. The requirements phase includes the pre-study phase and the feasibility study phase. The test phase involves the unit, system and acceptance tests. All these types of tests are included, since the data available did not separate between test types and they overlapped in terms of time.

3.2 Simulation Planning

This step included identifying factors that affect the quality of the developed software and the lead-time of the project. This was made through interviews with project staff and based on information in literature [5], [7].

The relevant project staff consisted in three persons with whom discussions were held continually during the entire study. Among the factors discovered during interviews, only those considered relevant to software development processes were selected. The identified factors are listed in Table 1. Discussions with concerned personnel pointed out the most important factors in respect to both quality and lead-time. The factors considered to affect quality and lead-time the most were chosen to be included in the influence diagrams, see Figure 2. Influence diagrams [12] for the requirements and the test phase were built to show how the chosen factors affect the lead-time and the software quality. Each factor's importance for each phase was considered together with the relationships between the factors. The influence diagram for the requirements phase is shown in Figure 2. The factors in the influence diagram are further explained below.

Table 1. Factors that affect quality and lead-time.

Number of personnel in the project	Amount of new market requirements
Level of personnel education	Amount of requirements changes
Level of personnel experience	Level of inadequate requirements
Level of personnel salary	Amount of review
Level of personnel turnover	Amount of rework
Communication complexity	Level of structure in the project organization
Geographical separation of the project	Standards that will be adhered to e.g. ISO and IEEE
Software and hardware resources	Amount of software functionality
Environment, e.g. temperature, light, ergonomics	Testing and correcting environment and tools
Amount of overtime and workload	Productivity
Level of schedule pressure	Amount of program documentation
Level of budget pressure	Level of reusable artefacts, e.g. code and documentation

- *Amount of functionality* is the estimated software functionality to be developed.
- *Amount of new market requirements* is a measure of the change in market expectations.
- *Amount of requirements changes* is a measure of the changes made in the requirements specifications.
- *Amount of review* involves reviewing requirements specifications.
- *Amount of rework* is the effort spent on reworking both new and inadequate requirements.
- *Communication complexity* is an effect in large project groups where an increasing number of participants increases the number of communication paths.
- *Level of inadequate requirements* is a measure of the requirements specification quality.
- *Level of personnel experience* is a measure of knowledge of the current domain.

- *Level of schedule pressure* is the effect of the project falling behind the time schedule.
- *Number of personnel* is the number of persons working with requirements specifications in the project.
- *Productivity* is a measure of produced specifications per hour and person.
- *Time in requirements phase* is the lead-time required to produce the requirements specifications in this project.

It is beyond the scope for this paper to present all details of the simulation model. In this paper the simulation model and related models, such as influence diagrams, are presented in some detail for the

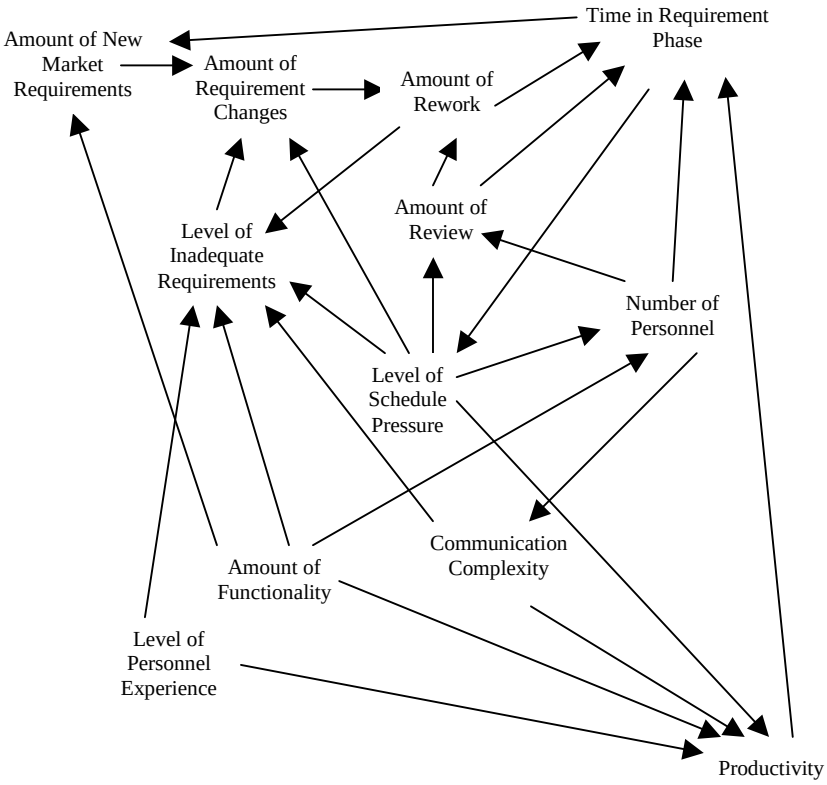


Figure 2. *Influence diagram*

requirements phase. The requirements phase is by its nature more intuitive and easy to understand than the test phase. For a presentation of details of the complete simulation model with all related models refer to [2]. For example, the influence diagram for the test phase is presented in [2] and not here.

At the same time as the influence diagrams were constructed, causal-loop diagrams were built to get a basic understanding of the feedback concepts. Causal-loop diagrams are often used in system dynamics to illustrate cause and effect relationships [1]. When examining these relationships isolated, they are usually very easy to understand. However, when they are combined into long chains of cause and effect, they can become complex. The causal-loop diagrams increase the understanding of these complex relations. Figure 3 illustrates how the schedule pressure affects the time spent in the requirements phase. An increased schedule pressure increases the error generation, due to a higher stress level. A high error density increases the amount of necessary rework and thereby increases the time in the requirements phase, which in turn increases the schedule pressure. At the same time, high schedule pressure increases the productivity because of its motivational role. Increased productivity decreases the time spent in the requirements phase, which in turn decreases the schedule pressure.

Information about the relationships between the factors in the causal-loop diagram is shown by adding an "O" or an "S" to the arrows. An "O" implies a change in the opposite direction, while an "S" implies a change in the same direction.

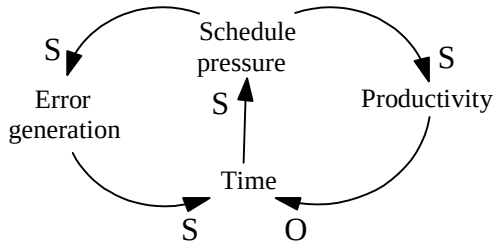


Figure 3. *Causal-loop diagram*

3.3 Simulation Operation

The simulation model was built based on the knowledge gained from creating influence diagrams and causal-loop diagrams. The idea behind the model of the requirements phase is based on a flow of tasks, from customer requirements to finished specifications. In the requirements phase there is a transformation from uncompleted to completed tasks by the production of specifications. A fraction of the specifications are not acceptable and needs to be taken care of in the rework loop, see Figure 4.

The test phase in the model is based on the same idea as the requirements phase and is built in a similar way. A flow of test cases is performed, a certain percentage of the functionality has to be corrected and retested, and the rest is supposed to be acceptable.

This basic model was built in the Powersim simulation tool [11] and further developed with help from the factors in the influence diagrams. Factors from the influence diagrams were added to the model in order to affect the levels and flows. The causal-loop diagrams were also considered during the development, to ensure that the model was adapted to systems thinking.

To avoid getting a too complex model, all of the factors in the influence diagrams were not included in the simulation model. Some factors were included indirectly in the parameters in the model. These can be extracted from the parameters and are thereby possible to affect from the user interface, for example the communication complexity which is included in the productivity. The construction was made step by step, by adding a few factors at a time and then running the simulation. The

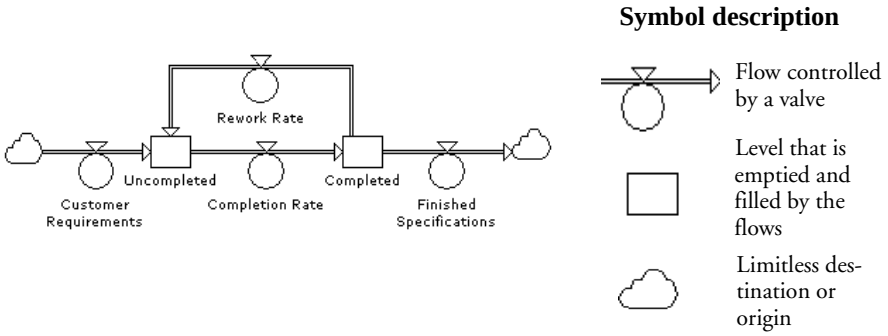


Figure 4. *Basic model of the requirements phase.*

values of the parameters were taken from project documentation except one that was taken from [7], *Amount of new market requirements*. This parameter was not available in project documentation but the value from [7] is an average from several software projects and was considered to be valid also for this project. Some values were estimated by iteration and verified by discussions with concerned personnel at the organization. The verification of the simulation model was made through checking that the amount of code that is used as an input to the model is the same as the output amount of code. The verification also included comparing the time in the simulation to the time according to the project documentation to ensure that the estimations were correct.

The final model for the requirements phase is seen in Appendix A. The flows in Figure 4 is the base of the final model, which is then further developed. To get a measure of the quality of the specifications, another flow was included, which counts the inadequate specifications. This measure affects the amount of defect code that is produced in the design and implementation phases which in turn affects the test phase. The design and implementation phases are in the simulation model modelled as a delay. A second flow is added to the basic model to terminate this phase and start the following phases.

The rest of the additions to the basic model can be described in four groups, where each group originates from the influence diagram.

- The first group, *Lines of code* and *Functionality,* describes the functionality of the code to be developed. This group controls the inflow to the phase.
- The second group is *Percentage, Effort* and *Duration.* The *Percentage* allocates a percentage of the planned total effort to the requirements phase and is controlled from the user interface. This makes it possible to study how the amount of resources in the requirements phase affects the lead-time and quality.
- The third group, *Productivity* and *Duration,* controls the completion rate of the specifications. The *Duration* also affects the amount of inadequate specifications because of the schedule pressure that might increase during the project's duration.
- The fourth group, *Amount of rework* and *Functionality*, decides how much of the specifications that needs to be reworked after the reviews.

Note that some factors are part of more than one group, since these affect more than one of the other factors.

4. Results from the Simulation

The final model was simulated to show how a relocation of resources to the different process phases affects the quality of the software products and the lead-time of the project. This model included both the requirements phase and the test phase. The model was run several times with different values of the percentage of the planned project effort, spent on the requirements phase. The results are given in precise figures but since there are a number of uncertainties they should be broadly interpreted. For example, the results are uncertain because of the difficulty in measuring the values of the included factors. It is the tendencies in the results that are important and not the exact figures.

The simulation runs indicate that the effort spent on the requirements phase has a noticeable effect on the lead-time of the project. The decrease in days, when increasing the effort in the requirements phase, arises from the increased specification accuracy. A more accurate specification facilitates the implementation and decreases the error generation and will result in a higher product quality from the start. This decreases the amount of necessary correction work and thereby shortens the time spent in the test phase. At a certain point the total lead-time will start to increase again because the time in the test phase stops decreasing while the time in the requirements phase continues to increase. The time in the test phase stops decreasing because there is always a certain amount of functionality that needs to be tested at a predetermined productivity. The number of days in Figure 5 is the total lead-time for the whole project.

In the same manner, the quality increases when increasing the effort in the requirements phase to a certain extent. The simulation runs indicate that the quality optimum appears in the same area as the lead-time optimum. The increase in quality originates from a higher specification accuracy, which is explained above. However, if too much effort is spent in the requirements phase, the quality will start to decrease again because there is less effort left for design, implementation and test tasks.

As a step in the verification of the results, they were compared to results in the software literature [7], [15]. This literature points at the

same magnitude of effort in the requirements phase for a successful project as the simulation results.

To summarize, the simulations indicate that there is an optimum for both the quality and the lead-time. If the effort in the requirements phase is lower than the optimal value, increasing it towards the optimum will result in increased quality of the developed software and decreased lead-time.

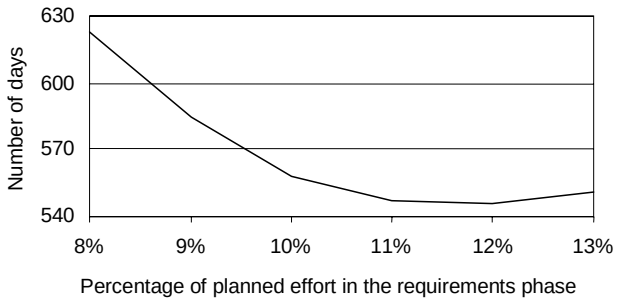


Figure 5. *Simulation results for the total lead-time.*

5. Discussion

One result of this study is a simulation model that visualizes different relations in a software development process. A simulation of this kind can contribute to enhancing the systems thinking in an organization. Thereby it is easier for the members of the organization to understand the relationships between the quality factors in the process.

The results from this kind of simulation shall not be interpreted precisely since there, of course, are a number of uncertainties. It is the tendencies and the behaviour in the results that are important and by changing the parameters in the model it is possible to get a picture of how the process mechanisms interact. This is a simplified model of the reality and therefore there are a number of sources of uncertainty. The included factors might not be the ones that affect the model the most, the assumed relations between the factors might not be correct and the values of the factors can be incorrectly estimated. However, the results, that there is an

optimum for the effort that is spent in the requirements phase, can be intuitively expected for many projects in software organizations.

A simulation of this kind can also be used to increase the motivation of the organization to work with quality issues and to increase the product quality early in the project.

One part of the knowledge gained from simulations is received in the model building process. The procedure to build the model forces the participants to communicate their mental models and to create a common image of the organization's direction.

To summarize, it seems to be feasible to build and use this kind of model for this kind of process. There are, however, a number of uncertainties which are important to take into account when the results are interpreted. This is a first model, based on one project, that needs to be further elaborated in order to obtain a model that can be applied on other projects. Thus, the model has not been empirically validated in real projects after it was developed. As far as the authors know, the model is not currently in use at Ericsson.

The impression after developing and getting feedback on the model is that it is uncertain whether most knowledge is gained by developing the model or using it. This is one of a number of issues that need to be further investigated in the area of software process simulation.

The model could either be used, for example by a project manager, by only changing the parameters, or it could be used by changing also the structure of the model, for example by adding or deleting factors and adding or deleting relationships between factors. It may be that users of the models need to understand the internal structure of the model and not only the interface to it. This would limit the choice of modelling techniques, and it would for example mean that models with an internal design, that is not easy to understand for the users of the models, would not be suitable in all cases.

Acknowledgement

The authors would like to thank Wladyslaw Bolanowski and Susanne S. Nilsson at Ericsson Mobile Communication AB for all their help with this study. This work is partly funded by the Swedish Agency for Innovation Systems (VINNOVA) under grant for Centre for Applied Software Research at Lund University (LUCAS).

6. References

[1] Abdel-Hamid, T., Madnick, S. E., *Software Project Dynamics: An Integrated Approach,* Prentice Hall, 1991.

[2] Andersson, C., Karlsson, L., "A System Dynamics Simulation Study of a Software Development Process", CODEN:LUTEDX(TETS-5419)/1-83/(2001)&local 3, Department of Communication Systems, Lund Institute of Technology, 2001.

[3] Banks, J., Carson, J. S., Nelson B. L., *Discrete-Event System Simulation,* Prentice Hall, 1996.

[4] Burke, S., "Radical Improvements Require Radical Actions: Simulating a High-Maturity Software Organization", Technical Report CMU/SEI-96-TR-024, Software Engineering Institute, Carnegie Mellon University, Pittsburg, USA, 1996.

[5] Fenton, N. E., Pfleeger, S., *Software Metrics: A Rigorous & Practical Approach,* International Thomson Computer Press, 1996.

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

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

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

[9] Martin, R., Raffo, D., "A Model of the Software Development Process Using both Continuous and Discrete Models", *International Journal of Software Process Improvement and Practice*, 5(2-3):147-157, 2000.

[10] Martin, R., Raffo, D., "Application of a Hybrid Process Simulation Model to a Software Development Project", *Proceedings of the Software Process Simulation Modeling Workshop*, 2000.

[11] Powersim Corporation, *www.powersim.com*, (last revisited 010903).

[12] Rus, I., Collofello, J. S., "Assessing the Impact of Defect Reduction Practices on Quality, Cost and Schedule", *Proceedings of the Software Process Simulation Modeling Workshop*, 2000.

[13] Senge, P. M., *The Fifth Discipline,* Random House Business Books, 1990.

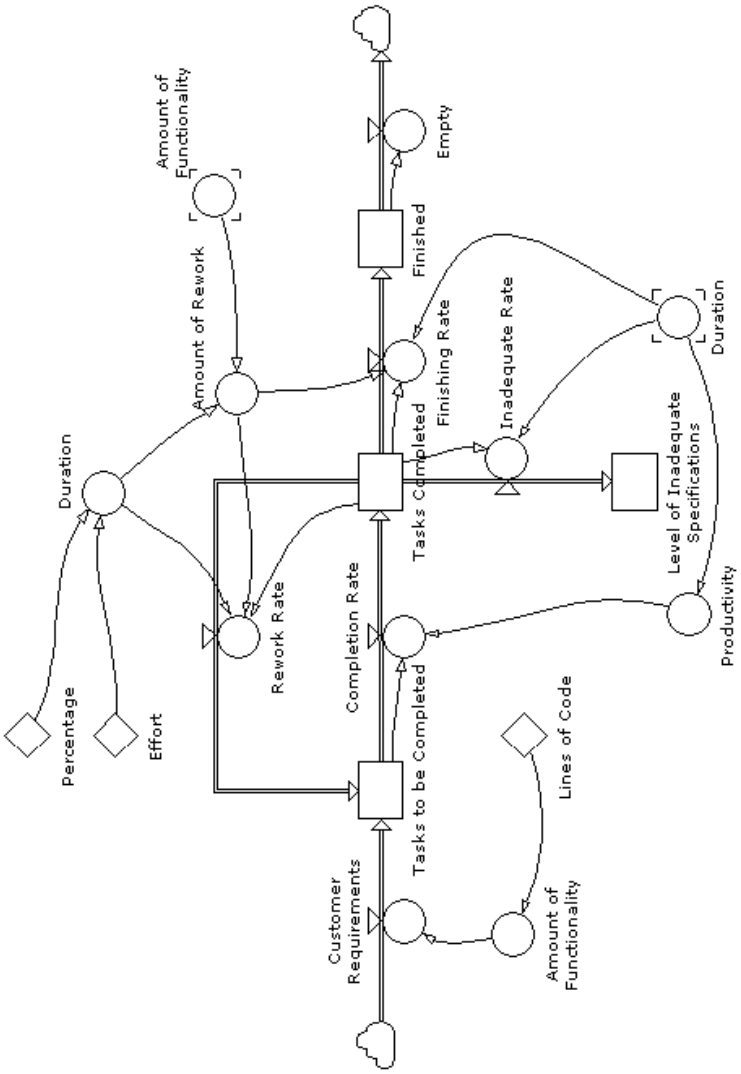
[14] Sommerville, I., *Software Engineering* (6th ed.), Addison-Wesley, 1996.

[15] Stewart, R. D., Wyskida, R. M., Johannes, J. D., *Cost Estimator's Reference Manual,* John Wiley & Sons Inc, 1995.

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

Appendix A

The simulation model of the requirements phase.



PAPER 3

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.

3

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

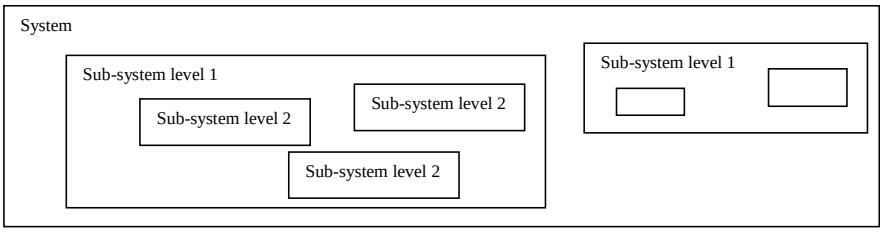


Figure 1. *The sub-system level 2 in the study in relation to the whole system.*

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.

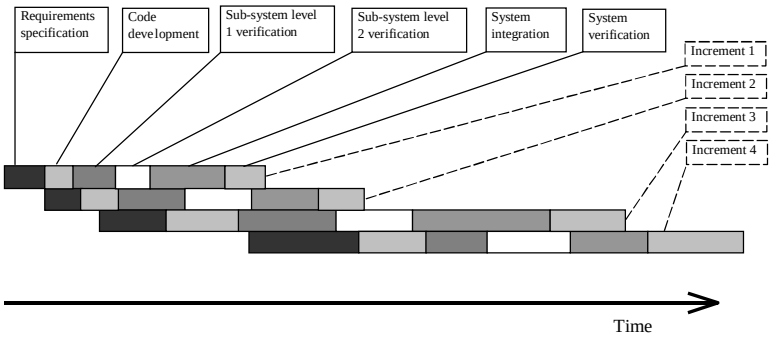


Figure 2. *An example of development phases and increments in an incremental development process.*

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.

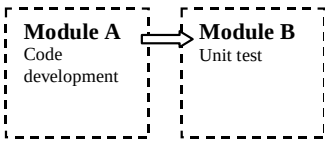


Figure 3. *The template model with one development phase and one test phase*

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.

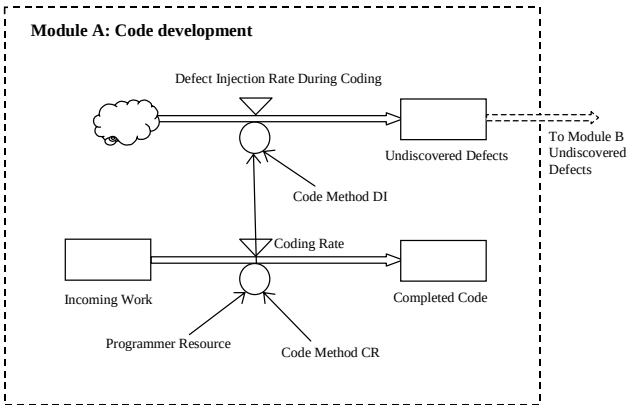


Figure 4. *The development module, module A, in the template model.*

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.

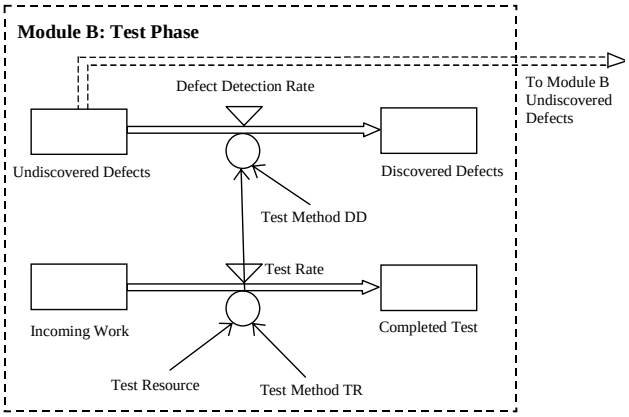


Figure 5. *The test module, module B, in the template model.*

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

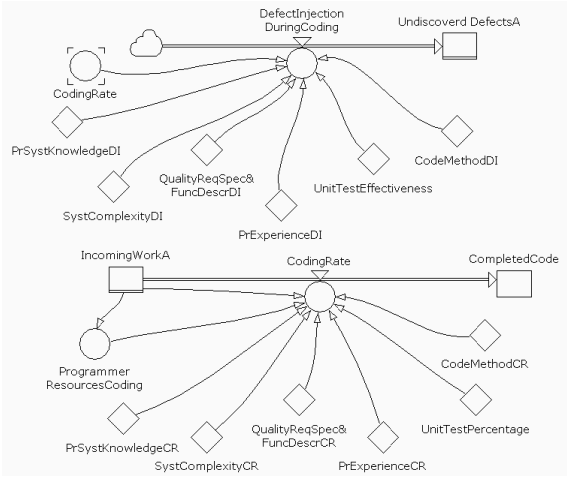


Figure 6. *Module A in the extended and adapted model in the industrial setting.*

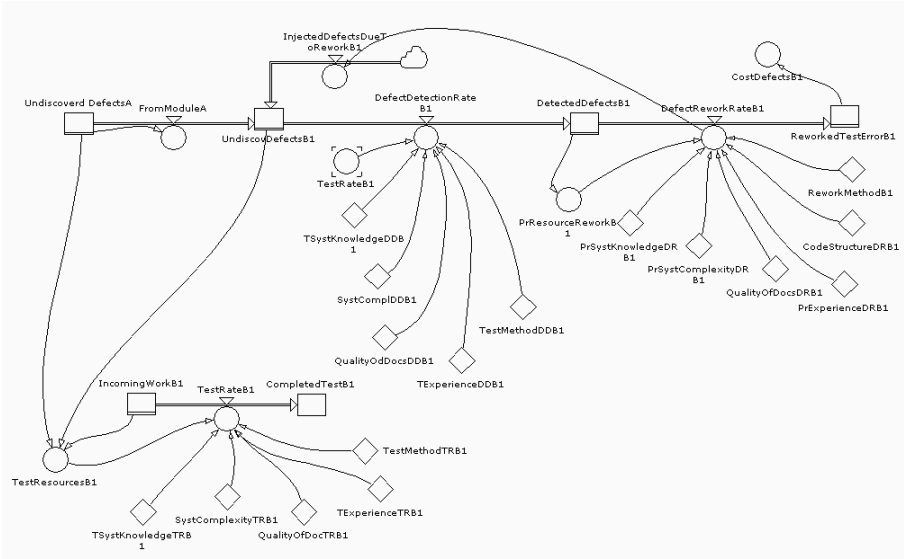


Figure 7. *Module B in the extended and adapted model in an industrial setting.*

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

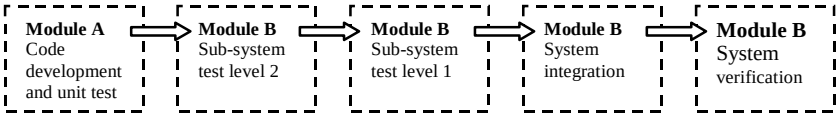


Figure 8. *Adapted and extended model in an industrial setting.*

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

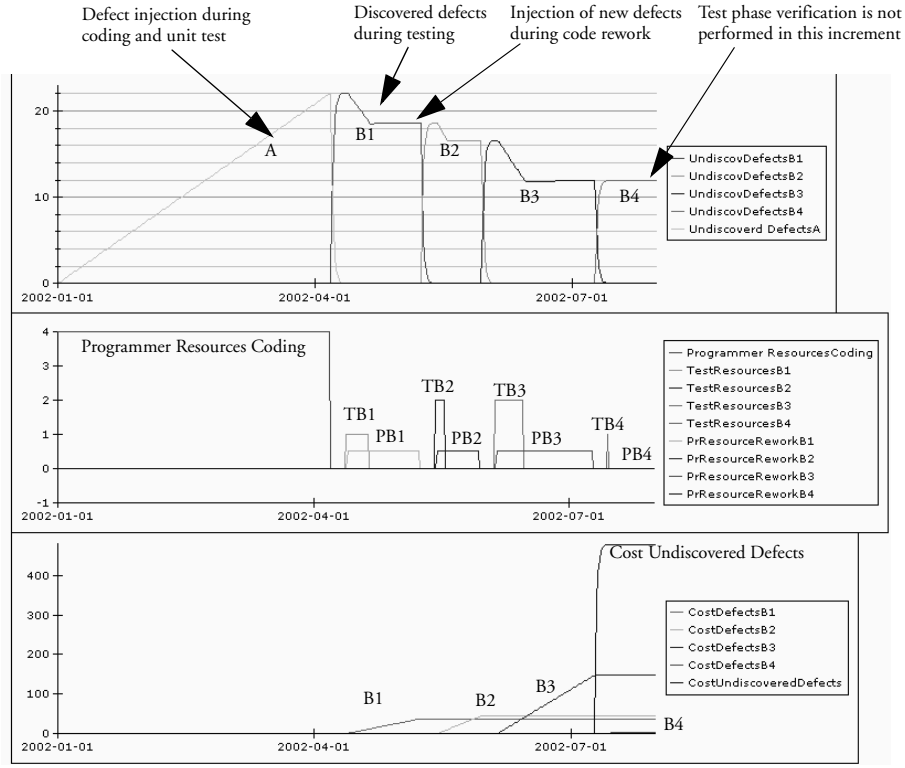


Figure 9. *Results from the simulation model in the industrial setting.*

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., Madnick, S., *Software Project Dynamics: An Integrated Approach*, Englewood Cliffs, New Jersey, Prentice Hall, 1991.

[2] Andersson, C., Karlsson, L., Nedstam, J., Höst, M., 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., 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., 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., 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., 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., 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., 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., 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 subsystems' 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 4

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.

4

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 the 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 are 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 what 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

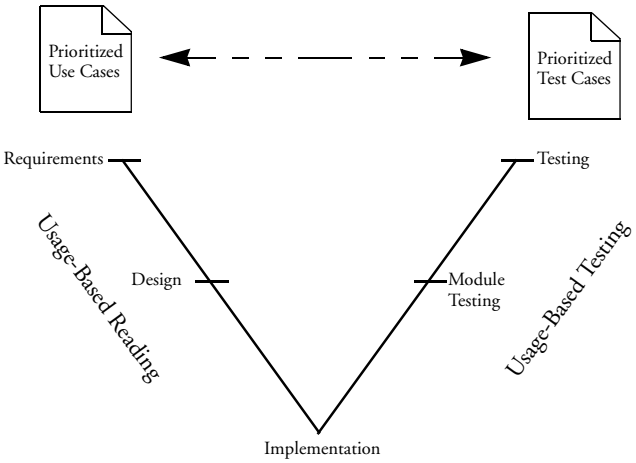


Figure 1. *The connection between the fault detection techniques compared in the empirical study.*

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.

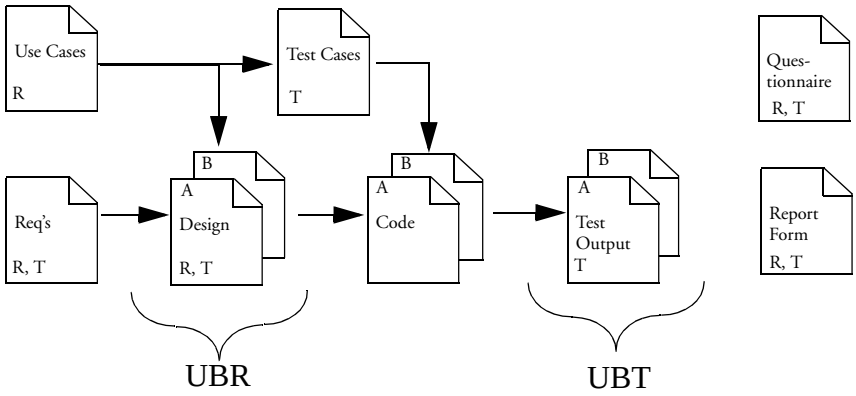


Figure 2. *Overview of the experiment artefacts. R means used by UBR and T means used by UBT. A and B refer to the different versions of the artefacts.*

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 . 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.

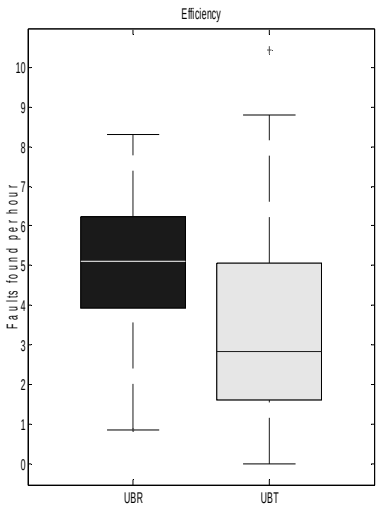


Figure 3. *Efficiency for UBR and UBT.*

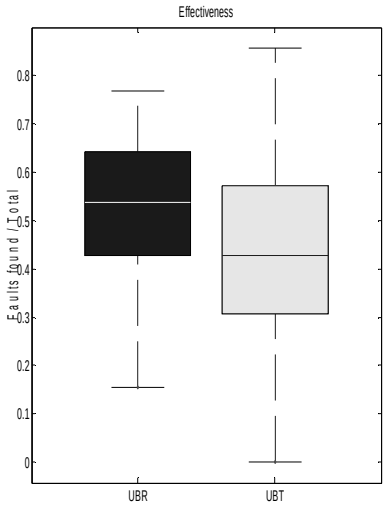


Figure 4. *Effectiveness for UBR and UBT.*

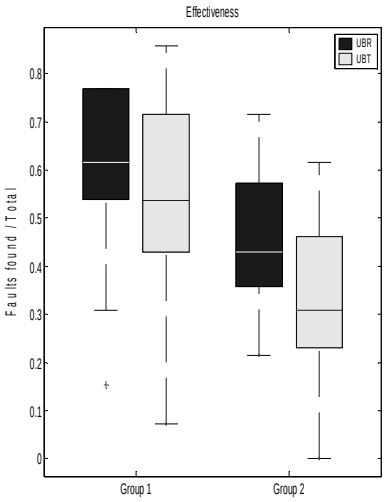


Figure 5. *Effectiveness for group 1 and 2. The black box plots show UBR, while the grey show UBT.*

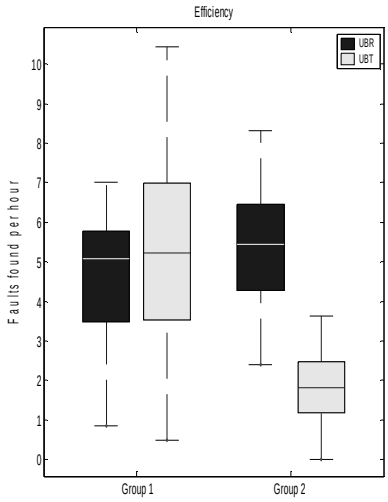


Figure 6. *Efficiency for group 1 and 2. The black box plots show UBR, while the grey show UBT.*

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

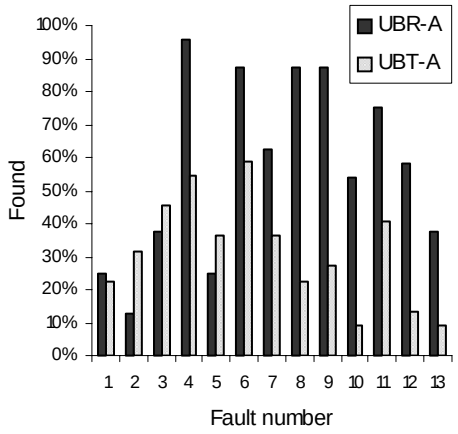


Figure 7. *Percentage of subjects detecting each fault in version A.*

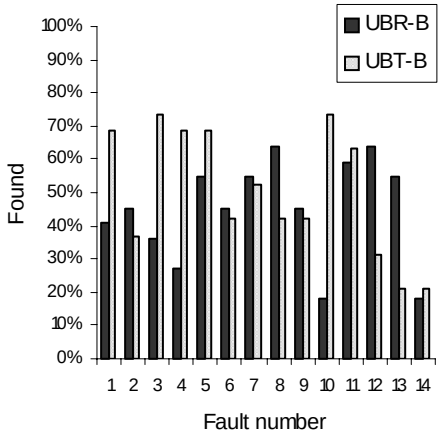


Figure 8. *Percentage of subjects detecting each fault in version B.*

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).

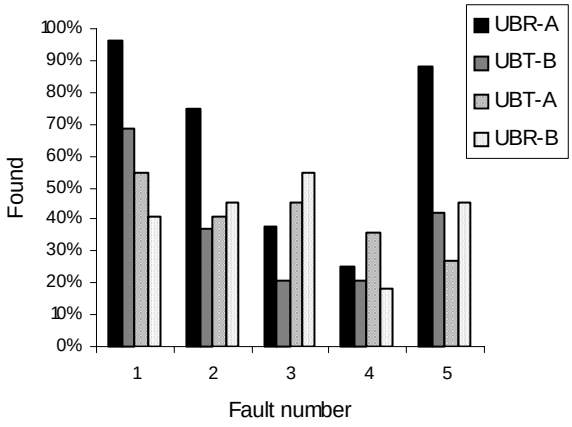


Figure 9. *Percentage of subjects finding the five faults that existed in both versions.*

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.

Reports on Communication Systems

101 **On Overload Control of SPC-systems**
Ulf Körner, Bengt Wallström, and Christian Nyberg, 1989.
CODEN: LUTEDX/TETS- -7133- -SE+80P

102 **Two Short Papers on Overload Control of Switching Nodes**
Christian Nyberg, Ulf Körner, and Bengt Wallström, 1990.
ISRN LUTEDX/TETS- -1010- -SE+32P

103 **Priorities in Circuit Switched Networks**
Åke Arvidsson, *Ph.D. thesis*, 1990.
ISRN LUTEDX/TETS- -1011- -SE+282P

104 **Estimations of Software Fault Content for Telecommunication Systems**
Bo Lennselius, *Lic. thesis*, 1990.
ISRN LUTEDX/TETS- -1012- -SE+76P

105 **Reusability of Software in Telecommunication Systems**
Anders Sixtensson, *Lic. thesis*, 1990.
ISRN LUTEDX/TETS- -1013- -SE+90P

106 **Software Reliability and Performance Modelling for Telecommunication Systems**
Claes Wohlin, *Ph.D. thesis*, 1991.
ISRN LUTEDX/TETS- -1014- -SE+288P

107 **Service Protection and Overflow in Circuit Switched Networks**
Lars Reneby, *Ph.D. thesis*, 1991.
ISRN LUTEDX/TETS- -1015- -SE+200P

108 **Queueing Models of the Window Flow Control Mechanism**
Lars Falk, *Lic. thesis*, 1991.
ISRN LUTEDX/TETS- -1016- -SE+78P

109 **On Efficiency and Optimality in Overload Control of SPC Systems**
Tobias Rydén, *Lic. thesis,* 1991.
ISRN LUTEDX/TETS- -1017- -SE+48P

110 **Enhancements of Communication Resources**
Johan M. Karlsson, *Ph.D. thesis*, 1992.
ISRN LUTEDX/TETS- -1018- -SE+132P

111 **On Overload Control in Telecommunication Systems**
Christian Nyberg, *Ph.D. thesis*, 1992.
ISRN LUTEDX/TETS- -1019- -SE+140P

112 **Black Box Specification Language for Software Systems**
Henrik Cosmo, *Lic. thesis*, 1994.
ISRN LUTEDX/TETS- -1020- -SE+104P

113 **Queueing Models of Window Flow Control and DQDB Analysis**
Lars Falk, *Ph.D. thesis*, 1995.
ISRN LUTEDX/TETS- -1021- -SE+145P

114 **End to End Transport Protocols over ATM**
Thomas Holmström, *Lic. thesis*, 1995.
ISRN LUTEDX/TETS- -1022- -SE+76P

115 **An Efficient Analysis of Service Interactions in Telecommunications**
Kristoffer Kimbler, *Lic. thesis*, 1995.
ISRN LUTEDX/TETS- -1023- -SE+90P

116 **Usage Specifications for Certification of Software Reliability**
Per Runeson, *Lic. thesis,* May 1996.
ISRN LUTEDX/TETS- -1024- -SE+136P

117 **Achieving an Early Software Reliability Estimate**
Anders Wesslén, *Lic. thesis*, May 1996.
ISRN LUTEDX/TETS- -1025- -SE+142P

118 **On Overload Control in Intelligent Networks**
Maria Kihl, *Lic. thesis,* June 1996.
ISRN LUTEDX/TETS- -1026- -SE+80P

119 **Overload Control in Distributed-Memory Systems**
Ulf Ahlfors, *Lic. thesis,* June 1996.
ISRN LUTEDX/TETS- -1027- -SE+120P

120 **Hierarchical Use Case Modelling for Requirements Engineering**
Björn Regnell, *Lic. thesis,* September 1996.
ISRN LUTEDX/TETS- -1028- -SE+178P

121 **Performance Analysis and Optimization via Simulation**
Anders Svensson, *Ph.D. thesis*, September 1996.
ISRN LUTEDX/TETS- -1029- -SE+96P

122 **On Network Oriented Overload Control in Intelligent Networks**
Lars Angelin, *Lic. thesis,* October 1996.
ISRN LUTEDX/TETS- -1030- -SE+130P

123 **Network Oriented Load Control in Intelligent Networks Based on Optimal Decisions**
Stefan Pettersson, *Lic. thesis,* October 1996.
ISRN LUTEDX/TETS- -1031- -SE+128P

124 **Impact Analysis in Software Process Improvement**
Martin Höst, *Lic. thesis,* December 1996.
ISRN LUTEDX/TETS- -1032- -SE+140P

125 **Towards Local Certifiability in Software Design**
Peter Molin, *Lic. thesis,* February 1997.
ISRN LUTEDX/TETS- -1033- -SE+132P

126 **Models for Estimation of Software Faults and Failures in Inspection and Test**
Per Runeson, *Ph.D. thesis*, January 1998.
ISRN LUTEDX/TETS- -1034- -SE+222P

127 **Reactive Congestion Control in ATM Networks**
Per Johansson, *Lic. thesis,* January 1998.
ISRN LUTEDX/TETS- -1035- -SE+138P

128 **Switch Performance and Mobility Aspects in ATM Networks**
Daniel Søbirk, *Lic. thesis,* June 1998.
ISRN LUTEDX/TETS- -1036- -SE+91P

129 **VPC Management in ATM Networks**
Sven-Olof Larsson, *Lic. thesis,* June 1998.
ISRN LUTEDX/TETS- -1037- -SE+65P

130 **On TCP/IP Traffic Modeling**
Pär Karlsson, *Lic. thesis,* February 1999.
ISRN LUTEDX/TETS- -1038- -SE+94P

131 **Overload Control Strategies for Distributed Communication Networks**
Maria Kihl, *Ph.D. thesis*, March 1999.
ISRN LUTEDX/TETS- -1039- -SE+158P

132 **Requirements Engineering with Use Cases – a Basis for Software Development**
Björn Regnell, *Ph.D. thesis*, April 1999.
ISRN LUTEDX/TETS- -1040- -SE+225P

133 **Utilisation of Historical Data for Controlling and Improving Software Development**
Magnus C. Ohlsson, *Lic. thesis,* May 1999.
ISRN LUTEDX/TETS- -1041- -SE+146P

134 **Early Evaluation of Software Process Change Proposals**
Martin Höst, *Ph.D. thesis*, June 1999.
ISRN LUTEDX/TETS- -1042- -SE+193P

135 **Improving Software Quality through Understanding and Early Estimations**
Anders Wesslén, *Ph.D. thesis*, June 1999.
ISRN LUTEDX/TETS- -1043- -SE+242P

136 **Performance Analysis of Bluetooth**
Niklas Johansson, *Lic. thesis,* March 2000.
ISRN LUTEDX/TETS- -1044- -SE+76P

137 **Controlling Software Quality through Inspections and Fault Content Estimations**
Thomas Thelin, *Lic. thesis,* May 2000
ISRN LUTEDX/TETS- -1045- -SE+146P

138 **On Fault Content Estimations Applied to Software Inspections and Testing**
Håkan Petersson, *Lic. thesis,* May 2000.
ISRN LUTEDX/TETS- -1046- -SE+144P

139 **Modeling and Evaluation of Internet Applications**
Ajit K. Jena, *Lic. thesis,* June 2000.
ISRN LUTEDX/TETS- -1047- -SE+121P

140 **Dynamic traffic Control in Multiservice Networks – Applications of Decision Models**
Ulf Ahlfors, *Ph.D. thesis*, October 2000.
ISRN LUTEDX/TETS- -1048- -SE+183P

141 **ATM Networks Performance – Charging and Wireless Protocols**
Torgny Holmberg, *Lic. thesis,* October 2000.
ISRN LUTEDX/TETS- -1049- -SE+104P

142 **Improving Product Quality through Effective Validation Methods**
Tomas Berling, *Lic. thesis,* December 2000.
ISRN LUTEDX/TETS- -1050- -SE+136P

143 **Controlling Fault-Prone Components for Software Evalution**
Magnus C. Ohlsson, *Ph.D. thesis*, June 2001.
ISRN LUTEDX/TETS- -1051- -SE+218P

144 **Performance of Distributed Information Systems**
Niklas Widell, *Lic. thesis,* February 2002.
ISRN LUTEDX/TETS- -1052- -SE+78P

145 **Quality Improvement in Software Platform Development**
Enrico Johansson, *Lic. thesis,* April 2002.
ISRN LUTEDX/TETS- -1053- -SE+112P

146 **Elicitation and Management of User Requirements in Market-Driven Software Development**
Johan Natt och Dag, *Lic. thesis,* June 2002.
ISRN LUTEDX/TETS- -1054- -SE+158P

147 **Supporting Software Inspections through Fault Content Estimation and Effectiveness Analysis**
Håkan Petersson, *Ph.D. thesis,* September 2002.
ISRN LUTEDX/TETS- -1055- -SE+237P

148 **Empirical Evaluations of Usage-Based Reading and Fault Content Estimation for Software Inspections**
Thomas Thelin, *Ph.D. thesis,* September 2002.
ISRN LUTEDX/TETS- -1056- -SE+210P

149 **Software Information Management in Requirements and Test Documentation**
Thomas Olsson, *Lic. thesis,* October 2002.
ISRN LUTEDX/TETS- -1057- -SE+122P

150 **Increasing Involvement and Acceptance in Software Process Improvement**
Daniel Karlström, *Lic. thesis,* November 2002.
ISRN LUTEDX/TETS- -1058- -SE+125P

151 **Changes to Processes and Architectures; Suggested, Implemented and Analyzed from a Project viewpoint**
Josef Nedstam, *Lic. thesis,* November 2002.
ISRN LUTEDX/TETS- -1059- -SE+124P

152 **Resource Management in Cellular Networks -Handover Prioritization and Load Balancing Procedures**
Roland Zander, *Lic. thesis,* March 2003.
ISRN LUTEDX/TETS- -1060- -SE+120P

153 **On Optimisation of Fair and Robust Backbone Networks**
Pål Nilsson, *Lic. thesis,* October 2003.
ISRN LUTEDX/TETS- -1061- -SE+116P

154 **Exploring the Software Verification and Validation Process with Focus on Efficient Fault Detection**
Carina Andersson, *Lic. thesis,* November 2003.
ISRN LUTEDX/TETS- -1062- -SE+134P