Monitoring and Implementing Early and Cost-Effective Software Fault Detection

Lars-Ola Damm

Blekinge Institute of Technology
School of Engineering

Blekinge Institute of Technology
Licentiate Series No. 2005:02

ISBN 91-7295-056-0

Printed in Sweden
Kaserntryckeriet AB
Karlskrona 2005

Practice is the best of all instructors
Publius Syrus (42 B.C)

This thesis is submitted to the Research Board at Blekinge Institute of Technology, in partial fulfillment of the requirements for the degree of Licentiate of Technology in Software Engineering.

Contact Information:

Lars-Ola Damm
Department of Systems and Software Engineering
School of Engineering
Blekinge Institute of Technology
P.O. Box 520
SE-372 25 Ronneby
SWEDEN

E-mail: lars-ola.damm@ericsson.com

Abstract

Avoidable rework constitutes a large part of development projects, i.e. 20-80 percent depending on the maturity of the organization and the complexity of the products. High amounts of avoidable rework commonly occur when having many faults left to correct in late stages of a project. In fact, research studies indicate that the cost of rework could be decreased by up to 30-50 percent by finding more faults earlier. However, since larger software systems have an almost infinite number of usage scenarios, trying to find most faults early through for example formal specifications and extensive inspections is very time-consuming. Therefore, such an approach is not cost-effective in products that do not have extremely high quality requirements. For example, in market-driven development, time-to-market is at least as important as quality. Further, some areas such as hardware dependent aspects of a product might not be possible to verify early through for example code reviews or unit tests. Therefore, in such environments, rework reduction is primarily about finding faults earlier to the extent it is cost-effective, i.e. find the right faults in the right phase.

Through a set of case studies at a department at Ericsson AB, this thesis investigates how to achieve early and cost-effective fault detection through improvements in the test process. The case studies include investigations on how to identify which improvements that are most beneficial to implement, possible solutions to the identified improvement areas, and approaches for how to follow-up implemented improvements.

The contributions of the thesis include a framework for component-level test automation and test-driven development. Additionally, the thesis provides methods for how to use fault statistics for identifying and monitoring test process improvements. In particular, we present results from applying methods that can quantify unnecessary fault costs and pinpointing which phases and activities to focus improvements on in order to achieve earlier and more cost-effective fault detection. The goal of the methods is to make organizations strive towards finding the right fault in the right test phase, which commonly is in early test phases. The developed methods were also used for evaluating the results of implementing the above-mentioned test framework at Ericsson AB. Finally, the thesis demonstrates how the implementation of such improvements can be continuously monitored to obtain rapid feedback on the status of defined goals. This was achieved through enhancements of previously applied fault analysis methods.

Acknowledgements

First and foremost, I would like to thank my supervisor *Lars Lundberg* for his support, especially for valuable feedback on the paper publications. I would also like to express my gratitude to my secondary advisor *Claes Wohlin* and my manager *Bengt Gustavsson* for making it possible for me to conduct this work. I also appreciate the support and guidance they have provided, e.g. Claes for various research related advice, and Bengt for enabling a good industrial research environment and for continuous feedback on ideas.

I would also like to thank all colleagues at *Ericsson AB* in Karlskrona that have taken part of or been affected by the research work. These include the members of the development projects that have been studied as a part of the research as well as line managers and the other members of the research project's steering group. Thanks for providing ideas and feedback, and for letting me interfere with the daily work when conducting the case studies. In particular, *David Olsson* has with a critical viewpoint provided a lot of valuable feedback, support and ideas. Without his help, the test automation framework presented in the thesis would probably not have been implemented. Further, I would like to thank *Johan Gardhage* for always being accessible to test ideas on and to get feedback from.

My colleagues in the research project BESQ, and the research groups SERL and PAARTS have also been very supportive. In particular, they have provided a scientific mindset and broadened my knowledge of software engineering research and practice. Especially, I would like to thank *Patrik Berander* for fruitful cooperation, not only in relation to the work presented in this thesis but for example also in university courses.

I also appreciate the help gotten from external persons, in particular *Johan Nilsson* for continuously giving feedback on papers, this especially to make sure that people outside the research environment also can understand them. Further, I am thankful for *Rikard Torkar's* help with resolving thesis formatting issues.

Finally, I would like to thank family and friends for putting up with me despite neglecting them when having a high work load.

This work was funded jointly by Ericsson AB and the Knowledge Foundation in Sweden under a research grant for the project "Blekinge - Engineering Software Qualities (BESQ)" (http://www.bth.se/besq).

Overview of Included Papers

Chapter 2 has not been written as a paper publication but is rather a summary of an initial case study evaluation that served as the starting point of this thesis (Damm 2002).
Chapter 3 is an extended version of a previously published paper. The original version was published in the *Proceedings of the 11th European Conference on Software Process Improvement*, Springer-Verlag, Trondheim, Norway, November 2004, pp. 138-149. The title of the chapter when published was ”Determining the Improvement Potential of a Software Development Organization through Fault Analysis: A Method and a Case Study”. The extended version was created based on an invitation for publication in a special issue of the *Journal of Software Process: Improvement and Practice*, Wiley InterScience.
Chapter 4 is scheduled for publication in *Electronic Notes in Theoretical Computer Science*, Elsevier. In this publication, the title of the paper will be ”Introducing Test Automation and Test-Driven Development: An Experience Report”. An earlier version of this paper has also been published at the *Conference on Software Engineering Research and Practice in Sweden*, Lund, Sweden, October, 2003.
Chapter 5 was in September 2004 submitted to the *Journal of Systems and Software*. At the time of this writing, no feedback has been obtained. The title of the submitted paper publication is ”Results from Introducing Component-Level Test Automation and Test-Driven Development”.
Chapter 6 is submitted to the *11th International Software Metrics Symposium*, IEEE, 2005. The submitted paper has the title ”Identification of Test Process Improvements by Combining ODC Triggers and Faults-Slip-Through”.
Chapter 7 is not yet submitted for publication.

Lars-Ola Damm is the main author of all chapters in this thesis. *Lars Lundberg* is a co-author of chapters 3-7 and *Claes Wohlin* is a co-author of Chapter 3. Additionally, *David Olsson* at Ericsson AB is a co-author of Chapter 4.

Contents

Chapter 1

Introduction

For most software development organizations, reducing time-to-market while still maintaining a high quality level is a key to market success (Rakitin 2001). During time-pressure, early quality assurance activities are commonly omitted hoping that the lead-time becomes shorter. However, since faults are cheaper to find and remove earlier in the development process (Boehm 1983), (Boehm and Basili 2001), (Shull et al. 2002), such actions result in increased verification costs and thereby also in increased development lead-time. Such an increase normally overshadows the benefits from omitting early quality assurance. In fact, the cost of rework commonly becomes a large portion of the projects, i.e. 20-80 percent depending on the maturity of the organization and the types of systems the organization develops (Boehm and Basili 2001), (Shull et al. 2002), (Veenendaal 2002). It has also been reported that the impact of defective software is estimated to be as much as almost 1 percent of the U.S. gross domestic product (Howles and Daniels 2003). Further, fewer faults in late test phases leads to improved predictability and thereby increased delivery precision since the software processes become more reliable when most of the faults are removed in earlier phases (Tanaka et al. 1995), (Rakitin 2001). Therefore, there is a growing interest in techniques that could find more faults earlier.

Although software development organizations in overall are aware of that faults are cheaper to find earlier, many still struggle with high rework costs (Boehm and Basili 2001), (Shull et al. 2002). In our experience, this problem is highly related to challenges of software process improvement in general, e.g. the conflict between short-term and long-term goals. ”When the customer is waving his cheque book at you, process issues have to go” (Baddoo and Hall 2003). Further, people easily become resistant because when under constant time-pressure, they do not have time to understand a change

and the benefits it will bring later (Baddoo and Hall 2003). Another contributing factor to failed quality-oriented improvement work is the way many programs are initiated, i.e. through heavy assessment and improvement programs such as CMM (Paulk et al. 1995), SPICE (El Emam et al. 1998), or Bootstrap (Card 1993). Such programs fail for many companies because they require significant long-term investments and because they assume that what works for one company works for another. However, this assumption is not true because there are no generally applicable solutions (Glass 2004), (Mathiassen et al. 2002).

Identification and implementation of smaller problem-based improvements is by many considered a more successful approach (Mathiassen et al. 2002), (Conradi and Fuggetta 2002), (Beecham and Hall 2003), (Eickelmann and Hayes 2004). However, companies using such an approach commonly identify a large number of problems of which all cannot be implemented at the same time. Therefore, the challenge is to prioritize the areas in order to know where to focus the improvement work (Wohlwend and Rosenbaum 1993). Without a proper decision support for selecting which problem areas to address, it is common that improvements are not implemented because organizations find them difficult to prioritize (Wohlwend and Rosenbaum 1993). To manage this problem, hard numbers on likely benefits from implementing suggested improvements are needed to make them possible to prioritize. Further, if the advantage of a suggested improvement can be supported with data, it becomes easier to convince people to make changes more quickly (Grady 1992).

When potential improvement areas are identified, organizations commonly know what need to be done to solve identified problems (Wohlwend and Rosenbaum 1993). However, traditional solutions are not always enough, new innovative solutions are sometimes needed. For example, as also have been confirmed in case studies presented in this thesis, deadline pressure commonly make developers not to conduct enough early quality assurance activities although they are well aware of their importance (Maximilien and Williams 2003). In such a case, it is not enough just to buy a new tool because the deadline pressure will most likely make them not use it anyway.

Nevertheless, the most challenging area in software process improvement seems to be the implementation of decided improvements. That is, the failure rate of process improvement implementation is reported to be about 70 percent (Ngwenyama and Nielsen 2003); turning assessment results into actions is when most organizations fail (Mathiassen et al. 2002). Therefore, practitioners want more guidance on how, not just what, to improve (Rainer and Hall 2002), (Niazi et al. 2005). This is especially needed in globally distributed software development where long distances and cultural differences make the implementation even harder.

A software development department at Ericsson AB wanted to address the above stated challenges and an initial assessment study determined that test-oriented improve-

ments would be most beneficial to focus on. Altogether, the above-stated challenges and the result of the conducted assessment study can be summarized into the following research questions to address:

1) Identification: *How should a software development organization make assessments in order to identify and prioritize test process improvements that achieve early and cost-effective fault detection?*

Besides identifying and prioritizing improvement areas, such assessments should include benchmark measures to compare future improvements against.

2) Solutions: *What test-oriented techniques are suitable for resolving identified improvement areas?*

Tailored from the IEEE standard, a test-oriented technique is in the context of this thesis defined as a technical or managerial procedures that aid in detecting faults earlier (IEEE 1990).

3) Implementation: *How can one make sure that test-oriented improvements are implemented successfully?*

This question regards how to specify goals that are easy to monitor and follow-up during and after projects, and the identification of other criteria that affect the success of an improvement effort. The primary challenge is to make sure that the improvement really is institutionalized instead of just fading out after the first implementation (Mathiassen et al. 2002). Due to the uncontrollable nature of industrial development projects, it is also a challenge to obtain quantitative data on the result of implemented improvements.

Through case studies at Ericsson AB, this thesis addresses the above-stated research questions with the purpose to determine how to achieve early and cost-effective fault detection through improvements in the test process. The remainder of this chapter describes how the research questions stated above were addressed together with an overview of related work and obtained results. Specifically, Section 1.1 provides an overview of previous work that is related to the areas addressed in this thesis. Thereafter, Section 1.2 provides an outline of this thesis and summarizes the contributions of the included chapters. Section 1.3 describes how the research has been conducted and the validity threats to the results. Then, Section 1.4 suggests areas within the scope of this thesis that need to be studied more thoroughly in further research. Finally, Section 1.5 concludes the work.

1.1 Concepts and Related Work

In relation to size, software is one of the most complex human constructs because no two parts are alike. If they are, they are made into one (Brooks 1974). Research and

development of techniques for making software development easier and faster have been going on for as long as software has existed. This section provides an overview of the software engineering concepts that the research presented in this thesis is based on. That is software testing and software process improvement.

1.1.1 Software Testing Concepts

Software has been tested from as early as software has been written because without testing, there is according to Graham (2001) no way of knowing whether the system will work or not before live use. Testing can be defined as *'a means of measuring or assessing the software to determine its quality'* (Graham 2001). The purpose of testing is two-fold: to give confidence in that the system works but at the same time to try to break it (Graham 2001).

Efficient Testing

Traditionally, test efficiency is measured by dividing the number of faults found in a test by the effort needed to perform the test (Pfleeger 2001). This can be compared to test effectiveness that only focuses on how many faults a technique or process finds without considering the costs of finding them (Pfleeger 2001). In the context of this thesis, an efficient test process verifies that a product has reached a sufficient quality level at the lowest cost.

In our experience, achieving efficient testing is dependent on a number of factors. They include using appropriate techniques for design and selection of test cases, having sufficient tool support, and having a test process that tests the different aspects of the product in the right order. An efficient test process verifies each product aspect in the test phase where it is easiest to test and the faults are cheapest to fix. Additionally, it avoids redundant testing. However, obtaining an efficient test process is not trivial because determining which techniques are suitable differs for each industrial context.

Cost of Testing

Testing does not in itself provide an added value to a product under development. Only the correction of the faults that were found during testing do. Therefore, companies want to minimize the cost of testing as much as possible. The cost of rework comprises a large proportion of software development projects, i.e. between 20-80 percent (Shull et al. 2002). Thus, the cost of faults strongly relates to the cost of testing. Figure 1.1 demonstrates how the cost of faults typically rises by development phase. For instance, the cost of finding and fixing faults is often 100 times more expensive after delivery

than during the design phase (Boehm and Basili 2001). The implication of such a cost curve is that the identification of software faults earlier in the development cycle is the quickest way to make development more productive (Groth 2004). In fact, the cost of rework could be reduced by up to 30-50 percent by finding more faults earlier (Boehm 1987). Therefore, employees should dare to delay the deliveries to the test department until the code has reached an adequate quality since high performing projects design more and debug less (DeMarco 1997). However, the differences in fault costs depend on the development practice used. That is, agile practitioners claim not to have such steep fault cost curves (Ambler 2004). The reason for this is a significantly reduced feedback loop, which is partly achieved through test-driven development (Ambler 2004). Finally, although succeeding in significantly reducing the cost of faults, this cannot eliminate the total test costs. The cost of designing and executing test cases always remain.

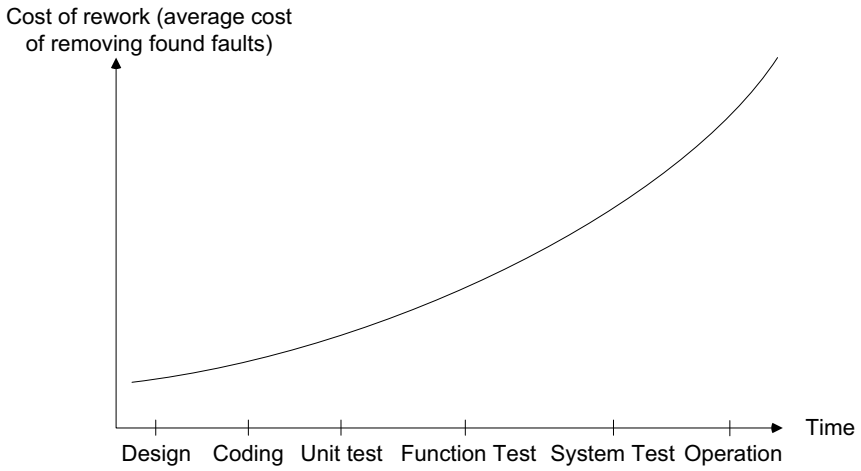


Figure 1.1: Cost of rework

Testability

The efficiency of a test process is highly dependent on how testable the product is. However, the practical meaning of testability is not obvious. The IEEE Standard Glossary of Software Engineering Terminology defines testability as:

The degree to which a system or component facilitates the establishment of test

criteria and the performance of tests to determine whether those criteria have been met (IEEE 1990)

Consequently, testability is a measure of how hard it is to satisfy a particular testing goal such as obtaining a certain level of test coverage or product quality. Two central characteristics of testability are how controllable and observable the system to test is (Freedman 1991). Further, a component with high testability can be characterized to have the following properties (Freedman 1991):

- The test sets are small and easily generated
- The test sets are non-redundant
- Test outputs are easily interpreted
- Software faults are easily locatable
- Inputs and outputs are consistent, i.e. given a certain input, only one output is possible

Both the product architecture and the test environment affect these attributes and must therefore be considered early to obtain adequate test efficiency. Testability cannot be achieved if it is an afterthought that follows design and coding (Beizer 1990). Finally, testability can also be seen as the test complexity of a product (Hicks et al. 1997), i.e. the testability is the degree of complexity from the viewpoint of the tester.

Test Levels

The test phases of a test process are commonly built on the underlying development process, i.e. each test phase verifies a corresponding design phase. This relationship is as illustrated in Figure 1.2 typically presented as a V-model (Watkins 2001). The implication of this is that for each design phase in a project, the designers/testers make a plan for what should be tested in the corresponding test phase before moving on to the next design phase. The contents of each test level differ a lot between different contexts. Different names are used for the same test levels and different contexts have different testing needs. However, in one way or another, most organizations perform the test activities. The purposes of the activities are as follows.

Module testing: Tests the basic functionality of code modules. The programmer who wrote the code normally performs this test activity (Graham 2001) and these tests are typically designed from the code structure. Since the tests focus on smaller chunks of code, it is easier to isolate the faults in such smaller modules (Patton 2001). This test level is commonly also referred to as unit testing, component testing, or basic testing.

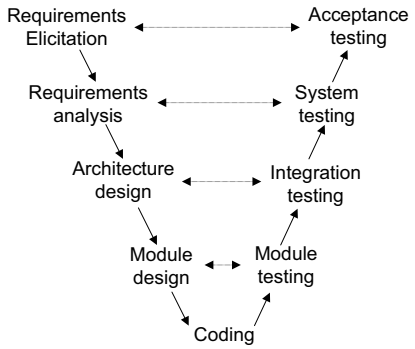


Figure 1.2: V-model for testing

Integration testing: When two or more tested modules are combined into a larger structure, integration testing looks for faults in the interfaces between the units and in the functions that could not be tested before but now can be executed in the merged units (Graham 2001). In some contexts, this test level is called function testing.

System testing: After integration testing is completed, system testing verifies the system as a whole. This phase looks for faults in all functional and non-functional requirements (Graham 2001). Depending on the scope of integration testing, function testing might be the main activity of this phase. In any case, the requirements that need to communicate with other systems and non-functional requirements are tested at this level.

Acceptance testing: When the system tests are completed and the system is about to be put into operation, the test department commonly conducts an acceptance test together with the customer. The purpose of the acceptance test is to give confidence in that the system works, rather than trying to find faults (Graham 2001). Acceptance testing is mostly performed in contractual developments to verify that the system satisfies the requirements agreed on. Acceptance testing is sometimes also integrated into the system testing phase.

In addition to these test levels, there is one vital test activity that is not considered as a standalone phase but rather is performed repeatedly within the other phases. That is, *regression testing*, which is applied after a module is modified or a new module is added to the system. The purpose of regression testing is to re-test the modified program in order to re-establish confidence that the program still perform according to its specification (White 2001). Due to its repetitive nature, regression testing is one of the most expensive activities performed in the software development cycle (Harrold

2000). In fact, some studies indicate that regression testing accounts for as much as one third of the total cost of a software system (Harrold 2000). To minimize this cost, regression testing relies heavily on efficient reuse of earlier created test cases and test scripts (Watkins 2001).

Test Strategies

A test strategy is the foundation for the test process, i.e. it states the overall purpose and how that purpose shall be fulfilled. Common contents of a test strategy is what test levels a test process should have and what each test level is expected to achieve. A test strategy does not state what concrete activities are needed to fulfill the goals. A test strategy could also state overall goals such as to find the faults as early as possible. The following paragraphs describe two typical strategic choices, which are independent of each other but still commonly are used together depending on the purpose of each particular test level.

Positive versus negative testing: The suitable test strategy is dependent on its purpose, i.e. if the tests should find faults (negative testing) or demonstrate that the software works (positive testing) (Watkins 2001). The most significant difference between positive and negative testing is the coverage that the techniques obtain. Positive testing only needs to assure that the system minimally works whereas negative testing commonly involves testing of special circumstances that are outside the strict scope of the requirements specification (Watkins 2001).

Black-box versus white-box testing: Most test techniques can be classified according to one of two approaches: black-box and white-box testing (Graham 2001). In black-box testing the software is perceived as a black box which is not possible to look into to see how the software operates (Patton 2001). In white-box testing the test cases can be designed according to the physical structure of the software. For example, if a function is performed with an “IF-THEN-ELSE” instruction, the test case can make sure that all possible alternatives are executed (Watkins 2001). Since white-box testing requires knowledge on how the software has been constructed, it is mostly applied during module testing when the developers that know how the code is structured perform the tests.

Test Techniques

Selecting an adequate set of test cases is a very important task for the testers. Otherwise, it might result in too much testing, too little testing or testing the wrong things (Patton 2001). Additionally, reducing the infinite possibilities to a manageable effective set and weighing the risks intelligently can save a lot of testing effort (Patton 2001).

The following list provides some of the possible techniques for test case selection (Graham 2001):

- Equivalence partitioning
- Boundary value partitioning
- Path testing
- Random testing
- State transition analysis
- Syntax testing (grammar-based testing)

Additionally, statistical testing is another commonly used approach for test case selection. The purpose of this technique is to test the software according to its operational behavior through operational profiles that describe the probability of different kinds of user input over time (Pfleeger 2001).

Further, software systems usually also have certain non-functional requirements, which require specific verification techniques. Below follows a few examples of such techniques (Watkins 2001):

- Performance testing
- Reliability testing
- Load testing
- Usability testing

1.1.2 Software Testing in this Thesis

This thesis includes solutions that address specific test areas, i.e. automated testing of software components and test-driven development as described in Chapter 4. This section provides an overview of concepts and related work within these areas.

Component Testing

Components can be defined in many different ways of which many recent definitions have in common that components should be possible to use as independent third-party components (Gao et al. 2003). However, due to the nature of the components in the studied context of this thesis, we define a component to be '*a module that encapsulates both data and functionality and is configurable through parameters in run-time*' (Harrold 2000). Thus, when discussing components in this thesis, no restrictions on whether the components should be possible to use as independent third-party products or not has been made.

The approach for testing component-based systems is closely related to object-oriented testing since a component preferably is structured around objects and interfaces and the structure of a component is very much alike that of an object (McGregor and Sykes 2001). Since components are run-time configurable, they can have different states and thereby several new test situations arise.

Component-based systems are commonly developed on a platform that at least contains a component manager and a standard way of communicating with other components. Typical commercial examples are CORBA, EJB and Microsoft COM (Gao et al. 2003). Such architectures are interesting from a testing point of view since in order to test the interfaces of such a component in isolation, the tests must be executed through the underlying platform. However, in our experience, this brings a large advantage because when all tests can be executed through a common interface, the test tool does only need to provide test interfaces against this common interface instead of one for each interface in every component. That is, the testability of such a component-based system is much higher than for systems having direct code-level communication. As further discussed in the next subsection, testability is especially important in automated testing.

Test Automation

Many think that automated testing is just the execution of test cases but in fact it involves three activities: creation, execution and evaluation (Poston 2005). Additionally, Fewster and Graham (1999) include other pre- and post- processing activities that need to be performed before and after executing the test cases (Fewster and Graham 1999). Such pre-processing activities include generation of customer records and product data (Fewster and Graham 1999). Further, post-processing activities analyze and sort the outputs to minimize the amount of manual work (Fewster and Graham 1999). However, the most important part of post-processing is the result comparison, i.e. the idea is that each test case is specified with an expected result that can be automatically com-

pared with the actual result after execution (Fewster and Graham 1999). The remainder of this section provides an overview of possible techniques and tools to consider when implementing test automation. Advantages and disadvantages of the different approaches are also discussed.

Code analysis tools: Pfleeger (2001) divides code analysis tools into two categories: static and dynamic analysis tools (Pfleeger 2001). Static code analysis tools can be seen as extensions to the compilers whereas dynamic code analysis tools monitor and report a program's behaviour when it is running. Typical examples of such tools are memory management monitors that for example identify memory leaks (Pfleeger 2001).

Test case generators: Test case generators can ensure that the test cases cover almost all possible situations for example by generating the test cases from the structure of the source code (Pfleeger 2001). Beizer lists a few approaches to test case generation (Beizer 1990):

- Structural test generators generate test cases from the structure of the source code. The problem with such generators is that they in the best case can provide a set of test cases that show that the program works as it was implemented. That is, tools that generate test cases from the code cannot find requirements and design faults.
- Data-flow-generators use the data-flow between software modules as a base for the test case generation. For example, they generate XML files to use as input data in the test cases.
- Functional generators are difficult to use because they require formal specifications that they can interpret. However, when working, they provide a more relevant test harness for the functionality of the system since they test what the system should do, i.e. as opposed to the structural test generators described above.

A good test case generator can save time during test design since it can generate several test cases fast and it has also the possibility to re-generate test cases in maintenance. However, when expected results need to be added manually or when the generator puts extra requirements on the design documentation, both development and maintenance costs will rise. When generating test input, it is very difficult to predict the desired outputs. Consequently, when the execution cannot be verified automatically, it leaves extra manual verification work for the testers (Beizer 1990).

Capture-and-replay tools: The basic strategy of capture-replay is that a tool records actions that testers have performed manually, e.g. mouse clicks and other GUI

events. The tool can then later re-execute the sequence of recorded events automatically. Capture-replay tools are simple to use (Beizer 1990), but according to several experiences not a good approach to test automation since the recorded test cases easily become very hard to maintain (Fewster and Graham 1999), (Kaner et al. 2002), (McGregor 2001). The main reason is that they are too tightly tied to details of user interfaces and configurations, e.g. one change in the user interface might require re-recording of 100 test scripts (Mosley and Posey 2002).

Scripting techniques: A common way to automate the test case execution is by developing test scripts according to a certain pattern. Script techniques provide a language for creating test cases and an environment for executing them (McGregor 2001). There exist several different approaches for developing test scripts:

- Linear scripts provide the simplest form of scripting and involves running a sequence of commands (Fewster and Graham 1999). This technique is for example useful for recording a series of keystrokes that are supposed to be used together repeatedly. However, no iterative, selective, or calling commands within the script are possible (Fewster and Graham 1999).

- Structured scripting is an extension to the one above and it can in addition to basic linear scripts handle iterative, selective, and calling commands. Such features make the scripts more adaptable and maintainable, especially when needing similar actions like for example performing variable changes within a loop are needed (Fewster and Graham 1999).

- Shared scripts can be used by more than one test case and these scripts are simply developed as scripts that are called from other scripts. This feature decreases the need for redundant script implementations and therefore simplifies the script code and increases the maintainability (Fewster and Graham 1999).

- Data-driven scripts provide another improvement for a script language by putting the test input in separate files instead of having it in the actual script (Fewster and Graham 1999). In data-driven testing, the data contained in the test data input file controls the flow and actions performed by the automated test script (Mosley and Posey 2002).

- Framework-driven scripts add another layer of functionality to the test environment where the idea is to isolate the software from the test scripts. A framework provides a shared function library that becomes basic commands in the tool's language (Kaner 1997), (Mosley and Posey 2002).

Frameworks provide the possibility to use wrappers and utility functions that can encapsulate commonly used functionality. Such mechanisms make maintenance easier because with well-defined wrappers and utility functions, an interface change only reflects the framework code instead of reflecting several test cases (McGregor 2001). Framework development and maintenance require dedicated recourses; however, such efforts can repeatedly pay for themselves since the quantity of test case code to write and maintain significantly decreases through the wrappers and utility functions (Kaner 1997). Data-driven testing is considered efficient since testers easily can run several test variants and because the data can be designed in parallel with the script code (Mosley and Posey 2002). Ultimately, the script technique to choose should be context dependent, i.e. the skills of the people together with the architecture of the product should determine which technique to choose (Kaner et al. 2002).

To summarize, different test automation techniques are feasible in different situations. In practice, it is common that test automation tools use a combination of techniques; for example, data-driven script techniques are sometimes combined with capture-replay. Further, it should be noted that automated test tools are not universal solvers; they are only beneficial when they are well-designed and used for appropriate tasks and environments (Fewster and Graham 1999). A tool does not teach how to test (Kaner et al. 2002).

Finally, the most important benefit of test automation can be obtained during regression testing. As previously stated, regression testing is a significant part of the total test cost and efficient reuse of test cases, e.g. through automatic re-execution, significantly decreases the regression test cost. Additionally, automated regression testing reduces the fault rates in operation because when the tests are cheaper to re-execute, more regression testing tend to be performed. In fact, related research reports that practicing automated regression testing at code check-in resulted in 36 percent reduction in fault rates (MacCormack et al. 2003). Another important aspect when considering test automation is the previously described testability attribute. That is, the success of test automation is highly dependent of having robust and common product interfaces that are easy to connect to test tools and that will not cause hundreds of test cases to fail upon an architecture change. The more testable the software is, the less effort developers and testers need to locate the faults (McGregor and Sykes 2001). In fact, testability might even be a better investment than test automation (Kaner et al. 2002).

Test-Driven Development

The concept of considering testing already during product design is far from new. For example, already 1983, Beizer stated that test design before coding is one of the most

effective ways to prevent bugs from occurring (Beizer 1983). The concept of Test-Driven Development (TDD) emerged as a part of the development practice eXtreme Programming (XP) (Beck 2003). However, among the practices included in XP, TDD is considered as one of few that has standalone benefits (Fraser et al. 2003).

The main difference between TDD and a typical test process is that in TDD, the developers write the tests before the code. A result of this is that the test cases drive the design of the product since it is the test cases that decide what is required of each unit (Beck 2003). '*The test cases can be seen as example-based specifications of the code*' (Madsen 2004). Therefore, TDD is not really a test technique (Beck 2003), (Cockburn 2002); it should preferably be considered a design technique. In short, a developer that uses traditional TDD works in the following way (Beck 2003):

1. Write the test case
2. Execute the test case and verify that it fails as expected
3. Implement code that makes the test case pass
4. Refactor the code if necessary

Nevertheless, the most obvious advantage of TDD is the same as for test automation in general, i.e. the possibility to do continuous quality assurance of the code. This gives both instant feedback to the developers about the state of their code and most likely, a significantly lower percentage of faults left to be found in later testing and at customer sites (Maximilien and Williams 2003). Further, with early quality assurance, a common problem with test automation is avoided. That is, when an organization introduces automated testing late in the development cycle, it becomes a catch for all faults just before delivery to the customer. The corrections of found faults lead to a spiral of testing and re-testing which delays the delivery of the product (Kehlenbeck 1997).

The main disadvantage with TDD is that in the worst case, the test cases duplicate the amount of code to write and maintain. However, this is the same problem as for all kinds of test automation (Hayes 1995). Nevertheless, to what extent the amount of code increases depends on the granularity of the test cases and what module level the test cases encapsulates, e.g. class level or component level. Nevertheless, TDD was foremost considered in our research because it can eliminate the risk for improperly conducted module testing. That is, during time-pressure, module testing tends to be neglected, but there is no reason not to module test the product when the executable test cases already are developed. Larger benefits with test automation come not only from repeating tests automatically, but also from testing use cases that were never executed at all before (Hayes 1995), (Mosley and Posey 2002).

Regarding the combination of automated component testing and TDD, little experience exists. TDD has as a part of XP been successfully used in several cases (Beck 2003), (Rasmusson 2004). However, the applicability of TDD has so far not been fully

demonstrated outside of that community. Teiniker et al. suggest a framework for component testing and TDD (Teiniker 2003). However, no practical results regarding the applicability of that concept exist. Further, as opposed to the solution suggested in this thesis, the framework described by Teiniker et al. focuses on model driven development and is intended for COTS (Commercial Of The Shelf) component development.

1.1.3 Software Process Improvement

Achieving earlier fault detection is not only about tools, a vital part of it is also to identify and implement improvements in the test process. Thereby, it is closely related to the notion of software process improvement, which especially during the 90's started gaining a lot of attention through quality assessment and improvement paradigms such as the ISO standard (ISO 1991) and the Capability Maturity Model (CMM) (Paulk et al. 1995). A major driving force was that if the development process has a high quality, the products that are developed with it also will (Whittaker and Voas 2002). However, even the best processes in the world can be misapplied (Voas 1997). Among the most important insights gained from using these models was that a process cannot be forced on people (Whittaker and Voas 2002) and that there is no one best way to develop software (Glass 2004), (Mathiassen et al. 2002). Since the realization that large process improvement frameworks are far from always the optimal solution, the process improvement paradigm can be characterized into working somewhere between two extremes, i.e. top-down versus bottom-up process improvement.

Top-down versus Bottom-up

The above-mentioned improvement paradigms are sometimes called top-down approaches to process improvement since they provide a set of best practices that are to be adhered by all organizations. From feedback of earlier applications of ISO and CMM, enhanced versions of these top-down frameworks have been developed. For example, SPICE (ISO/IEC 15504), which is focused on process assessments (El Emam et al. 1998) and Bootstrap which is another top-down process maturity framework developed by a set of European companies as an adaptation of CMM (Card 1993). Further, CMM has been tailored into sub-versions such as SW-CMM, which is adapted for software development (SW-CMM). Recently, an integration of existing CMM variants has also been gathered into a model called CMMI (Capability Maturity Model Integration) (CMMI 2002). Work has also been done to tailor CMM to test process improvement, i.e. the Test Maturity Model (TMM) (Veenendaal 2002). Here CMM has been adapted to what the test process should achieve for each maturity level.

The opposite of applying a top-down approach is the bottom-up approach where improvements are identified and implemented locally in a problem-based fashion (Jakobsen 1998). The bottom-up approach is sometimes also referred to as an inductive approach since it is based on a thorough understanding of the current situation (El Emam and Madhavji 1999). A typical bottom-up approach is the Quality Improvement Paradigm (QIP), where a six step improvement cycle guides an organization through continuous improvements based on QIP (Basili and Green 1994). From defined problem-based improvements, QIP sets measurable goals to follow-up after the implementation. Therefore, the Goal-Question-Metric (GQM) paradigm (Basili 1992), which is centered around goal-oriented metrics, is commonly used as a part of QIP (Basili and Green 1994). Since problem-based improvements occur spontaneously in the grassroots of several organizations, several other more pragmatic approaches to bottom-up improvements exist (Jakobsen 1998), (Mathiassen et al. 2002). Identifying problems and then improve against them can be achieved without using a formal framework.

The basic motivation for using the bottom-up approach instead of the top-down approach is that process improvements should focus on problems in the current process instead of trying to follow what some consider to be best practices (Beecham and Hall 2003), (Glass 2004), (Jakobsen 1998), (Mathiassen et al. 2002). That is, just because a technique works well in one context does not mean that it also will in another (Glass 2004). Another advantage with problem-based improvements is that the improvement work becomes more focused, i.e. one should identify a few areas of improvement and focus on those (Humphrey 2002). Nevertheless, this does not mean that the top-down approaches will disappear. Quality assessment frameworks appear to be useful for example in sectors developing pharmaceutical and automotive software. Further, these frameworks could guide immature companies that do not have sufficient knowledge of what aspects of their processes to improve. However, in these cases they should be considered as recipes instead of blueprints, which historically has been the case (Aaen 2003).

Fault Based Approaches

Problem-based improvements can in practice be managed in several ways depending on what aspects of the process to address. Therefore, when as in this thesis the objective is to address when faults are detected, a fault-based improvement approach is preferable. Fault based approaches have possibilities to achieve higher quality (Bassin et al. 1998), (Biehl 2004) or might aim at decreasing fault removal costs during development (Chillarege and Prasad 2002), (Hevner 1997). Analyzing fault reports is an easy way to identify improvement issues and triggers focused improvement actions.

Further, the results are highly visible through hard fault data (Mathiassen et al. 2002). In fact, some claim that fault analysis is the most promising approach to software process improvement (Grady 1992).

Basically, there are two main approaches to fault analysis (Bhandari et al. 1993):

Causal analysis: A commonly used approach for identifying activities that need improvements is classification of faults from their causes, e.g. root cause analysis (Leszak et al. 2000). Although root cause analysis can provide valuable information about what types of faults the process is not good at preventing or removing, the technique is cost intensive and therefore not easy to apply on larger populations of faults. Further, root cause analysis can only suggest improvements during design and coding.

Goal-oriented analysis: This approach is strongly related to the earlier described GQM paradigm. The approach captures data trends on faults and compares those trends to specified improvement goals. Examples of captured trends are number of faults in relation to product size or to classify faults according to certain classification schemes.

The most widespread method for number of faults in relation to product size is Six Sigma (Biehl 2004), (Mast 2004). Six Sigma is strongly goal-driven since it is centered around a measure with the goal value of Six Sigma. That is, Six Sigma is when a product does not produce more than 3.6 faults per million opportunities (Whittaker and Voas 2002). The main advantage of the method is that it is purely customer-driven (Mast 2004) whereas it has been criticized for the vague measurement definition, i.e. how should 'a million opportunities' be measured?

A commonly used classification scheme approach for fault analysis is Orthogonal Defect Classification (ODC) (Chillarege et al. 1992). Here, a defect has the same meaning as a fault, i.e. an anomaly that causes a failure (IEEE 1988). Thus, the terms fault and defect are used as synonyms within the work presented in this thesis. ODC focuses on two types of classifications for obtaining process feedback, i.e. fault type and fault trigger classification. ODC fault type classification can provide feedback on the development process whereas ODC fault trigger classification can provide feedback on the test process (Chillarege et al. 1992). Fault type classification can provide development feedback by categorizing faults into cause related categories such as assignment, checking, interface, and timing (Chillarege et al. 1992). However, as we also have experienced during our research work, this classification scheme has proven to be hard to apply in practice. That is, it was hard to make unambiguous classifications since it was not obvious which category a fault should belong to (Henningsson and Wohlin 2004). Further, connecting ODC fault types to improvement actions is not always obvious. That is, the focus of such fault classifications must be on tying process improvements to the removal of specific types of faults (Grady 1992).

The other ODC classification method focuses on fault triggers, i.e. a test activity that makes a fault surface. An ODC trigger scheme divides faults into categories such

as concurrency, coverage, variation, workload, recovery, and configuration (Chillarege and Prasad 2002). ODC fault trigger classification has in our experience appeared to be more promising than ODC fault type classification since test related activities are easier to separate from each other and could also more easily be connected to a certain improvement actions. Further, a trigger scheme is easy to use since the cause of each fault needs not to be determined; only the testers' fault observations are needed. Commonly, ODC fault triggers are used for identifying discrepancies between the types of faults the customers find and those that are found during testing. Through such analysis, test improvements can be made to ensure that customers find fewer faults in consecutive releases (Bassin et al. 1998). Dividing trigger distributions after which phases the faults were found in can visualize what types of faults different phases find. If this distribution over phases is not in accordance with the test strategy, improvements could be targeted against the trigger types that occurred too frequently in relation to what was expected.

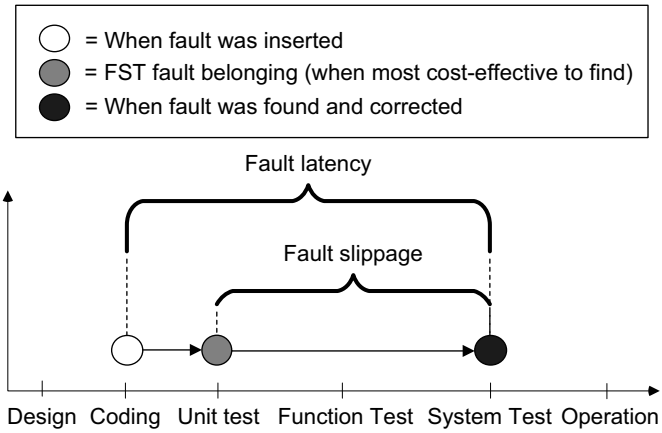


Figure 1.3: Example of Fault latency and Faults-slip-through

Other possible fault classification methods include classification by priority, phase found, and fault latency. As illustrated in Figure 1.3, the latter works against the goal that faults should be found in the same phase as they were introduced (Grady 1992) or that faults should be found in the earliest possible phase (Berling and Thelin 2003). These measures are similar to one of the primary measures used in this thesis, i.e. faults-slip-through, which is defined as whether a fault slipped through the phase where it should have been found. The main difference is that faults-slip-through does not

consider phase inserted but instead when it would have been most cost-effective to find the fault. Thus, the primary difference is that when defining FST, the test efficiency is in focus; the fault insertion stage is not considered. The FST measure requires more up-front work than the fault latency concept because which types of faults that should be found in which phase must be predefined. However, the fault latency concept is not feasible for test process improvements because most faults are inserted during design and coding activities, which results in a very high FST that is more or less impossible to decrease.

Process Implementation

As stated in the introduction of this chapter, the failure rate of process improvement implementation is reported to be about 70 percent (Ngwenyama and Nielsen 2003). Therefore, it is not surprising that practitioners want more guidance on how, not just what to improve (Rainer and Hall 2002), (Niazi et al. 2005). Much of the failure is blamed on the above-described top-down frameworks since they do not consider that since software process improvement is creative, feedback-driven, and adaptive; the concepts of evolution, feedback, and human control are of particular importance for successful process improvement (Gray and Smith 1998). The process improvement frameworks lack an effective strategy to successfully implement their standards or models (Niazi et al. 2005). However, several studies have been conducted to determine characteristics of successful and failed software process improvement attempts. In a study that assembled results from several previous research studies and also conducted a survey among practitioners, the following success factors and barriers were identified as most important (Niazi et al. 2005).

Success factors:

- Senior management commitment
- Staff involvement
- Training and mentoring
- Time and Resources

Barriers:

- Lack of resources
- Organizational politics

- Lack of support
- Time pressure
- Inexperienced staff
- Lack of formal methodology
- Lack of awareness

Most of these success factors and barriers are probably well-known to anyone that has conducted process improvement in practice. However, the two last barriers require an explanation. As also acknowledged in the beginning of this section, a lack of formal methodology concerns guidance regarding how to implement process improvements. Further, awareness of process improvements are important to get long-term support by managers and practitioners to conduct process improvements (Niazi et al. 2005).

Another barrier not mentioned in this study but frequently in others is 'resistance to change' (Baddoo and Hall 2003). In the list above, this barrier is however strongly related to staff involvement and time pressure. That is, process initiatives that do not involve practitioners are de-motivating and unlikely to be supported, and if they do not have time to understand the benefit a change will give, they are resistant to the change (Baddoo and Hall 2003). A success factor frequently mentioned in other studies is that measuring the effect of improvements increases the likelihood of success (Dybå 2002), (Rainer and Hall 2003). Metrics for measuring process improvements are further discussed in the next subsection.

From the results of an analysis of several process improvement initiatives at a department at Ericsson AB in Gothenburg, Sweden, another important success factor was identified (Borjesson and Mathiassen 2004). That is, the likelihood of implementation success increases significantly when the improvement is implemented iteratively over a longer period of time. The main reason for this was that the first iteration of an implemented change results in a chaos that causes resistance. However, the situation stabilizes within a few iterations and if the chaos phase is passed, the implementation is more likely to succeed (Borjesson and Mathiassen 2004).

Process Metrics

Empirical research commonly uses one of three main metrics areas, i.e. oriented at products, resources, or processes (Fenton and Pfleeger 1997). Since this thesis studies the development process, the area of process metrics is highly related to the context of this thesis. Further, as stated in the previous section, software metrics is considered as an important driver for process improvement programs (Gopal et al. 2002), (Offen and

Jeffery 1997). A large reason for this is that you cannot control what you cannot measure (DeMarco and Lister 1987), (Fenton and Pfleeger 1997), (Gilb 1988), (Maxwell and Kusters 2000). However, establishing software metrics in an organization is not trivial. In fact, one study reported a mortality rate for metrics programs to be about 80 percent (Rubin 1991). To increase the likelihood of success, Gopal has gathered a set of factors that are important when initiating metrics programs (Gopal et al. 2002):

- Appropriate and timely communication of metrics results are critical in order to be used
- It is important to build credibility around metrics for the success of software process improvement
- The effort required to collect metrics should not add significantly to the organizations workload
- The success of a metrics program is dependent on the impact it has on decision making, not the longevity of it
- People who are trained or exposed to metrics are likely to use them more
- Automated tool support has a positive influence on metrics success

Additionally, related research states that organizations with successful measurement programs actively respond to the obtained measurement data (McQuaid and Dekkers 2004). Further, in metrics programs, it is important to accept the current situation and then improve on it. When tying reward systems to metrics programs, accurate data tend to be withheld (McQuaid and Dekkers 2004). Finally, defining clear objectives is crucial for success. Metrics should only be a part of an overall process improvement strategy, otherwise they provide little value (Rakitin 2001).

1.2 Outline and Contribution of the Thesis

This section describes an outline of the chapters included in this thesis together with their contributions. The first part below describes the overall content and contribution followed by a more detailed chapter-by-chapter overview.

Figure 1.4 visualizes the outline of the chapters of this thesis, i.e. it illustrates a time perspective on the chapters and how they relate to each other. In the figure, the initial case study assessment presented in Chapter 2 served as the starting point of the research. That is, the case study determined where to focus the improvement work. After that, Chapter 3 further supports the assessment results through a benchmark measure

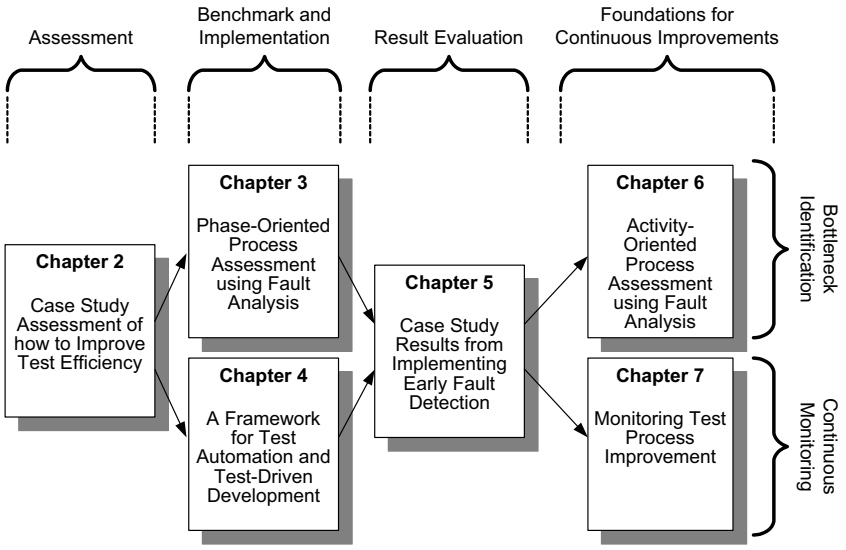


Figure 1.4: Chapter Outline

and Chapter 4 presents the implementation of the framework that was suggested to address the most important improvement area identified in Chapter 2. Chapter 5 uses the method and benchmark measure in Chapter 3 when evaluating the implementation result of the framework described in Chapter 4. After having the framework in place, the last two chapters address two new identified needs, i.e. more fine-grained improvement identification (Chapter 6) and a possibility to monitor implemented improvements in a fast and quantitative way (Chapter 7).

The overall contribution of this thesis was obtained through the results of a series of case studies conducted in an industrial setting, i.e. the contents of the chapters outlined above. The common denominator of the case studies is that they all contribute to the overall objective of the conducted research, i.e. to minimize the test lead-time through early and cost-effective fault detection. Specifically, the thesis has two major contributions that also have been validated in practice:

- A framework for test automation and test-driven development
- A set of methods for identifying and evaluating test process improvements

Further, each conducted case study contributes to at least one of the three research

questions that were listed in the first section of this chapter, i.e. *identification*, *solutions*, and *implementation*. Besides summarizing contents and contributions, the description of each chapter below states which of the research questions that were addressed in that chapter.

1.2.1 Chapter 2

This chapter provides a summary of results from a case study that served as input to the research project that resulted in this thesis. The purpose of the case study was to evaluate the development process of a software development department at Ericsson AB and from the results of the evaluation suggest improvements that would decrease the test lead-time. The main contributions of the chapter are an overview of the state of a software development organization's processes and an identification of where in the process improvements were needed. Most importantly, the case study assessment provided the starting point for further research, i.e. it determined that further research and practical improvements should focus on achieving earlier fault detection through improvements in the test process. The specific contributions sorted by research question are:

Identification: The case study assessment identified a number of potential improvement areas that should be addressed:

- Many faults were more expensive to correct than needed
- There was a lack of tool support for developers and testers
- Deadline pressure caused neglected unit testing
- Insufficient training on test tools decreased the degree of usage

Solutions: A literature review identified four state-of-the-art techniques that could aid in test efficiency improvements, i.e. Orthogonal Defect Classification (Chillarege and Prasad 2002), risk-based testing (Amland 2000), (Veenendaal 2002), automated testing, and techniques for component testing. From the case study assessment and literature review, a total of ten candidate improvements were suggested. Of these candidates, concrete improvement proposals were made for two of them because they were considered especially feasible to implement. The most important improvement comprised enhancements of the module testing level.

1.2.2 Chapter 3

The main objective of the case study presented in this chapter was to investigate how fault statistics could be used for determining the improvement potential of different phases/activities in the software development process. The chapter provides a solution based on a measure called faults-slip-through, i.e. the measure tells which faults that should have been found in earlier phases. From the measure, the improvement potential of different parts of the development process is estimated by calculating the cost of the faults that slipped through the phase where they *should* have been found. The usefulness of the method was demonstrated by applying it on two completed development projects. The results determined that the implementation phase had the largest improvement potential since it caused the largest faults-slip-through cost to later phases, i.e. 85 and 86 percent of the total improvement potential in the two studied projects.

The main contribution of this chapter is the faults-slip-through based method since it can determine in what phases to focus improvement efforts and what cost savings such improvements could give. Further, the practical application of the method quantified the potential benefits of finding more faults earlier, e.g. that the fault slippage cost could be decreased by up to about 85 percent depending on the additional costs required to capture these faults in the implementation phase instead. The faults-slip-through measure also appeared to be useful in test strategy definitions and improvements. That is, specifying when different fault types should be found triggered test strategy improvements. Regarding specific contributions in relation to the research questions of this thesis, all contributions of this chapter addressed the research question named *identification*.

1.2.3 Chapter 4

This chapter presents an approach to software component-level testing that in a cost-effective way can move fault detection earlier in the development process. The approach was based on the evaluation results presented in Chapter 2.

The approach comprised a framework based on component-level test automation where the test cases are written before the code, i.e. an alternative approach to Test-Driven Development (TDD) (Beck 2003). The implemented approach differs from how TDD is used in the development practice Extreme Programming (XP) in that the tests are written for components exchanging XML data instead of writing tests for every method in every class. This chapter describes the implemented test automation tool, how test-driven development was implemented with the tool, and experiences from the implementation.

The overall contribution of this chapter is the technical description of the frame-

work for component-level test automation and TDD and lessons learned from introducing the framework in two software development projects at Ericsson AB. Therefore, the main contributions of this chapter concern *solutions.* However, the lessons learned also address the research question on *implementation*.

1.2.4 Chapter 5

Chapter 4 describes a concept for component-level test automation and TDD. However, the chapter does not provide quantitative results from the practical application of the concept. The purpose of this chapter is twofold. First, a method for evaluating the result of such an introduction needed to be identified. This method should then be used for determining the result of the framework application. Based on the faults-slip-through measure described in Chapter 3, this chapter proposes a method for how to obtain quantitative results from an improvement such as the one to evaluate. The result of the practical evaluation was that the implementation of the new concept had significant improvement rates in form of decreased faults-slip-through rates (5-30 percent improvement) and decreased fault costs, i.e. the avoidable fault cost was on average about 2-3 times less than before. It was concluded that these savings by far exceeded the costs of performing additional testing in earlier phases.

This chapter mainly contributes to the thesis through the quantitative evaluation results that demonstrated that the concept described in Chapter 4 was successfully applied in practise, i.e. the *solution* decreased the fault costs significantly. Further, regarding *implementation*, the chapter proposes and demonstrates the applicability of a method for evaluating fault-directed improvements in an industrial setting. For example, the method could determine to what extent the obtained results occurred due to the implemented concept, i.e. by filtering away effects of eventual other changes in other test phases.

1.2.5 Chapter 6

This chapter is based on the hypothesis that Faults-Slip-Through (FST), the fault analysis method described in Chapter 3, can be combined with ODC fault trigger classification (Chillarege et al. 1992) to provide useful input to test process improvements. Specifically, the chapter proposes a method for how a combination of these techniques can be used to identify which test activities that could be improved and in which test phase the improvement for each activity is needed. Results from applying the method on a case study project resulted in a set of prioritized improvements to implement in a consecutive project.

The contribution of this chapter is the *identification* method for combining ODC fault trigger classification and faults-slip-through to identify test process improvements. The method identifies which fault triggers that have high frequencies in relation to if they were faults-slip-though from a certain phase and in relation to their average fault cost. From such an analysis, the fault triggers that are most beneficial to address can be identified.

1.2.6 Chapter 7

Chapter 7 describes an approach for how to use FST for monitoring implemented test process improvements in software development. The purpose of the method is to obtain early feedback from improvement goals that have been defined for a project. The method was applied in two projects, of which one was developed at an offshore location.

The contribution of this chapter regards *implementation* of test process improvements. Specifically, the case studies demonstrated the applicability of the implemented method as it provided early feedback on the test process performance in the projects. Further, the case studies resulted in a number of lessons learned, e.g. that cultural differences affected the results.

1.3 Research Methodology

The previous section described the contents and contributions of this thesis. The purpose of this section is to describe the methods used for obtaining these contents and contributions. The following outline below presents how this method description is presented.

The next section provides an overview of possible research methods to use for achieving the goals of the research. After that, the sections 1.3.2 and 1.3.3 describe how the research was conducted. Finally, Section 1.3.4 discusses the validity of the results.

1.3.1 Research Methods

Research approaches

Since the research objective of this thesis is to study and improve practice, empirical methods are most suitable for achieving that. The list below describes the methods that were possible to use for this type of research.

Survey: Surveys are commonly seen as research-in-the-large since the typical strategy is to send out a questionnaire to a large number of widely distributed people (Wohlin et al. 2003). Therefore, surveys usually contain fixed questions that provide quantitative answers that are easier to analyze. Typically, surveys intend to generalize from a sample to a population (Creswell 2003).

Experiment: On the contrary to surveys, experiments are sometimes referred to as research-in-the-small since they typically address a limited scope and are performed in a laboratory setting (Wohlin et al. 2003). Experiments are highly controlled because their purposes are to test the impact of a treatment on an outcome when at the same time controlling other factors that might intervene (Creswell 2003).

Case study: Case studies are considered as research-in-the-typical because their objectives are to study a real industrial project (Wohlin et al. 2003). Case studies are normally easier to plan but the results are difficult to generalize and harder to interpret (Wohlin et al. 2003).

Action research: A related area to case studies is action research in which improvement and active involvement in practice are central parts (Robson 2002). Action research differs from the other research methods in that its purpose is to influence change (Robson 2002). Typically, action research is conducted in cycles, i.e. plan, act, observe, and reflect (Pourkomeylian 2002). The main benefit with action research is that it is possible to monitor the result of a change while actively changing it (Martella et al. 1999). A drawback is that it requires more efforts than other research methods (Martella et al. 1999).

Post-mortem analysis: Post-mortem analysis is another research approach that commonly is used in conjunction with case studies. The basic idea with this approach is to capture knowledge and experience from a case or activity after its completion (Wohlin et al. 2003). Post-mortem analysis has the same scope as case studies but has also a similarity to surveys in that it studies the past (Wohlin et al. 2003).

Which method is best suited for a particular situation depends on several factors such as the type of study, the purpose of the study, and available resources. Experiments are good when the objective is to compare two situations in a controlled environment, e.g. compare two techniques (Wohlin et al. 2003). However, when implementing an actual improvement in an industrial setting the case study approach is most likely best suited because they can avoid scale-up problems (Wohlin et al. 2003). Action research could also be a feasible alternative in such a situation. Surveys are typically conducted to get a snapshot view of the current situation or to collect attitudes towards a newly introduces technique (Wohlin et al. 2003). Further, post-mortem analysis mainly targets larger projects to learn from success or recovery from failure. However, the method is very flexible (Wohlin et al. 2003). Regarding the type of results to obtain, all methods expect

experiments are both qualitative and quantitative; experiments are purely quantitative (Creswell 2003). Finally, the four methods can be combined and should therefore be seen as complementary and not competing (Wohlin et al. 2003).

Data Collection Methods

Data collection is a central and indispensable part of empirical research (Robson 2002). The primary choice when collecting data is if qualitative or quantitative data is of interest. That is, should the research provide numerical or non-numerical results or perhaps both (Robson 2002). Quantitative methods are good for studying causes and effect relationships between variables (Creswell 2003). However, quantitative methods need predefined questions and therefore, qualitative methods are better for observational studies trying to understand a situation (Creswell 2003). Further, in many cases, a combination of qualitative and quantitative methods is preferable for example because then the validity of the results can be strengthened through data triangulation (Creswell 2003). The choice of research methods to use strongly affects the type of data to be obtained. Nevertheless, as described below, both qualitative and quantitative data can be obtained in several ways.

Interviews and Questionnaires: In empirical research, the inquiry of individuals through interviews or questionnaires is a natural source of information. Interviews are typically used in case studies and there are two extreme approaches for how to conduct them. Unstructured open-ended interviews are used when the researcher does not really know what to ask for or want to ensure that the case study is of a qualitative nature, i.e. the questions do not strictly control the interviews (Creswell 2003). On the other hand, standardized open-ended interviews are used when more control is preferred. That is, to simplify comparisons of responses, the respondents are asked the same questions in the same order (Martella et al. 1999). Typically, interviews provide input for consecutive quantitative questionnaires that can confirm what was indicated in the interviews (Creswell 2003). Questionnaires are also an integral part of surveys, partly because quantitative data from larger surveys is easier to analyze (Creswell 2003). Further, questionnaires can sometimes also support experiments since experiments are purely quantitative (Wohlin et al. 2003).

Unobtrusive Data Collection: Unobtrusive data collection focuses on collecting and studying traces that people or projects leave behind (Robson 2002). Typical traces are project documentation such as specifications, project reports, and fault reports. The large advantage of unobtrusive measures is that they are produced in their real setting without being interfered by the research context (Robson 2002). That is, they are not subject to validity threats such as biases or the quality of the research design. The main disadvantages are that the data might be incomplete and that it is hard to assess causal

relationships (Robson 2002). Unobtrusive data collection is commonly used in both case studies and post-mortem analysis. Further, action research commonly also obtain some unobtrusive measures from direct observations and documents (Pourkomeylian 2002).

1.3.2 Research Approach and Environment

Research Environment

The case studies presented in this thesis were conducted at a software development department at Ericsson AB in Sweden. The department runs several projects in parallel and consecutively. The projects develop releases of products that are employed as network services on top of base stations systems. Each project on average lasts about 1-1.5 years and has on average about 50 participants. Typically, the products have mostly non-human interfaces, i.e. protocol-based communication with other systems in the mobile network. Human interfaces are only used for configuration and administration purposes. Further, the development projects have three main test steps:
Basic Test: Isolated tests of individual units or components.
Function Test: Verifies the functional requirements of the product in a simulated environment.
System Test: In this test phase, focus is put on system-level requirements such as performance, robustness, and redundancy. Additionally, System Test includes verification against external systems using test plants configured with the hardware and software to be incorporated with the product in operation.

The product development organization consists of a set of units of which the software design and test units have been studied during the research. The design units are responsible of detailed design, implementation, and Basic Test whereas the test units conduct Function and System Test. Finally, the products are maintained by a separate maintenance organization utilizing designers and testers from the design and test units.

Utilized Research Approaches

The overall purpose of the research was to apply and investigate the behavior of existing techniques in practice, not to invent new ones from scratch. Nevertheless, investigating the practical applicability of promising techniques triggered new inventions in form of refinements of the applied techniques. The findings were obtained through the following research approaches.

The research was mainly performed through a number of case studies, commonly in the form of two or more case studies applying the same method or implementing

the same improvement in different projects and products. Additionally, post-mortem analysis was used mainly for post-analysis of fault reports and to some extent action research was also applied due to direct involvement in the studied projects. Besides fault reports, data was collected from interviews, questionnaires, and project documentation. That is, a combination of qualitative and quantitative methods was used. The next section discusses the implications of utilizing such approaches.

Applied Research

The research results presented in this thesis have been obtained through applied research at the above described software development department. This has not only involved to conduct specific case studies but has also included continuous observation of, and participation in, the work conducted at the department. The researcher's role can be expressed to have been a 'practitioner-researcher', i.e. to hold down a job of some particular area and is at the same time involved in carrying out systematic research that is of relevance to the job (Robson 2002). The main advantage of conducting research in this way is that the research is carried out in a realistic context. That is, if software engineering research is not based on problems identified in industry and proposed solution are not tested in an industrial environment, the research does not provide any real benefit to industry (Glass 1994). A disadvantage is reduced individual freedom in the selection of research studies, which however is balanced by an increasing likelihood of implementation (Robson 2002). Another disadvantage with applied research is that industrial settings tend to differ in a way that can threaten generalizability (Wohlin et al. 2000). Nevertheless, applied research can also bring other benefits such as increased knowledge transfer between industry and academia, e.g. industry can propose relevant research questions to academia and ideas and methods from academia can be transferred to industry.

1.3.3 Research Process

This section describes how the improvement work has evolved. That is, the work process for identifying research questions and then finding answers to them. The conducted steps as described below were not predefined but rather evolved from emerging needs. The chapters that provide the corresponding results are stated within the parenthesis.

1. Performed an assessment study to determine in what areas improvements were needed (Chapter 2).

2. Implementation of a solution according to the assessment result in step one (Chapter 4).

3. Identified a need for quantitative evaluation of improvements. This led to an identification of a method (Chapter 3) and application of the method on the implemented solution (Chapter5).

4. Needed a fine-grained identification method for identifying further improvement opportunities. This led to the identification of a new method (Chapter 6).

5. Realized a need for continuous follow-up of implemented improvements, including goal monitoring in distributed development (Chapter 7).

1.3.4 Validity of the Results

Results from research conducted in an industrial environment have a high validity in the sense that the research environment is realistic. However, other aspects might affect the validity of the results since an industrial environment is harder to control than for example a controlled experiment. This section discusses a number of threats to the validity of the results obtained in this thesis.

Internal validity concerns how well the study design allows the researchers to draw conclusions from causes and effects, i.e. causality. For example, there might be factors that affect the dependent variables (e.g. fault distributions) without the researchers knowing about it (Wohlin et al. 2003). Internal validity is in general a large threat to case studies since the industrial environment changes over time. That is, it cannot be controlled to the same extent as for example experiments. In our research, the risk for unknown factors affecting the results was fairly low since the researcher has been able to observe the projects closely and was thereby more likely to notice factors or events that could affect the results. Further, during the research period, the personnel turnover was low, i.e. each new project release had more or less the same personnel as the previous release. Thereby, the likelihood that the 'people factor' affected the results was low.

Conclusion validity concerns whether it is possible to draw correct conclusions from the results, e.g. reliability of the results (Wohlin et al. 2003). Three typical threats to conclusion validity are reactivity, participant bias and researcher bias (Robson 2002). Reactivity means that a researcher's presence might affect the setting and thereby the behavior of the people involved. A common example of respondent bias is to withhold information, and researcher bias includes assumptions and preconceptions that might for example affect the questions asked or the interpretations made during data analysis (Robson 2002). Both reactivity and participant bias were in the conducted

research minimized through prolonged involvement. However, being an active participant could however also increase the researcher bias. The main approach used for addressing this threat was through member checking where the participants reviewed the obtained results (Robson 2002). Applying member checking also decreased the risk for subjectivity, i.e. objectivity can be obtained through inter-subjective agreement (Robson 2002). Additionally, unobtrusive measures were also used to support and confirm the obtained results.

External validity concerns whether the results are generalizable or not (Robson 2002). In case studies, obtained results are in overall not fully generalizable since they have been obtained in the context of the studied environment. As earlier stated, conducting applied research in an industrial setting threatens the generalizability of the results since each industrial setting has its own context. Therefore, the results of this thesis are not fully generalizable since they are dependent on the context where they were applied. That is, the results are generalizable within the context of the studied department whereas the generalizability decreases more and more the less alike the considered context is. Nevertheless, the methods presented and applied in this thesis are in overall generalizable since they are more or less independent of the context. However, the usefulness of some of the methods and techniques is likely to be larger in some contexts than others. For example, the methods that are based on fault analysis are primarily aimed for organizations developing consecutive releases of a product or that develop several similar products using the same process. The likelihood of success is also larger if the target organizations have rather well established processes in place, e.g. test automation requires structured test processes (Kaner et al. 2002) and fault analysis is naturally easier when having common fault tracking procedures and fault report documentation.

1.4 Further Work

This thesis presents several techniques for how to improve the test process. However, the techniques have more or less been developed and applied in isolation. Further work should attempt to gather these techniques into some kind of common framework or toolbox. For example, an appropriate order of usage should be defined. Plausibly, this could be obtained through integration with an existing framework for continuous improvement, e.g. the Six Sigma paradigm that includes an improvement cycle consisting of the steps define-measure-analyze-improve-control (SixSigma 2005).

For each of the research questions investigated in this thesis, the following candidate additions to current work are suggested.

1.4.1 Identification

Replicated studies on the applicability of the techniques presented in thesis are needed. That is, some open issues still remain, such as for which contexts and purposes they are suitable and not. Further elaboration on suitable combinations of methods is also of interest.

Module-directed fault classification is another area that would contribute to the existing set of methods. That is, studies from different environments over several years have shown that 80 percent of the faults in a system comes from 20 percent of the modules (Boehm and Basili 2001). Further research should determine how existing methods for module classification should be combined with the faults-slip-through and fault-trigger concepts to obtain useful information.

1.4.2 Solutions

The most interesting improvement in this area would be to make replicated implementations of the solution presented in Chapter 4. However, since the test automation tool is dependent on the proprietary platform the components are running on, this is not possible. However, it would be of interest to make replicated studies of the underlying approach. That is, to apply component-level test-driven development in other environments.

Another possible area to study is how the introduction of automated function tests affects the fault distributions. That is, a research study could determine whether the amount of fault slippages to subsequent phases decreases and the fault trigger distributions change or not from introducing the test automation tool.

1.4.3 Implementation

The need for research in this area appears to be large, i.e. studies indicate that up to two thirds of the managers involved in software process improvement need guidance on how to implement improvement activities rather than what activities to implement (Niazi et al. 2005). Within the research of this thesis, possible contributions to this area include enhancements of the method introduced in Chapter 7. For example, as for the faults-slip-through measure, it might be possible to also include fault trigger distributions when conducting early and continuous monitoring of implemented improvements.

Organizations have different objectives, e.g. reduce development cost versus reduce customer quality or they have different improvement opportunities, e.g. phase-oriented, activity-oriented or module-oriented improvements. Therefore, further research should

also investigate how to select which improvement approaches and in which order to implement them. One promising method that could aid in such selections is the previously mentioned Goal-Question-Metric (GQM) paradigm (Basili 1992).

Finally, the results in Chapter 7 indicated that process improvement cannot be managed in the exact same way when having the development organization distributed to geographically dispersed locations, i.e. some considerations need to be made. The methods described in this thesis need further studies in distributed environments in order to fully understand the impact of applying them there.

1.5 Conclusions

This thesis presents the results of a set of case studies conducted in an industrial environment. The overall objective of the research has been to decrease the test lead-time through early and cost-effective fault detection. The obtained results include a framework for test automation and test-driven development and a set of fault-directed methods for identifying and evaluating test process improvements.

The main conclusions of this thesis are:

- There is a large potential in moving fault detection earlier in the process
- Working in a test-driven way increases the test focus in earlier phases and thereby moves fault detection earlier in the process
- Classifying faults into a matrix divided after when faults where found respectively when they should have been found provides information that can be used in several ways to identify and monitor improvements in the test process
- Tools are an integral part of most improvements aimed at the test process, e.g. both in concrete solutions, and as support when identifying and monitoring improvements

Chapter 2

Case Study Assessment of how to Improve Test Efficiency

Lars-Ola Damm

2.1 Introduction

This chapter presents a case study evaluation that was the starting point of the research presented in this thesis. Besides summarizing the case study findings, the chapter includes a retrospective discussion on what has happened to the identified improvement areas since then. The chapter is outlined as follows. The remainder of this section describes the case study background. After that, Section 2.2 describes the method used for obtaining the case study results presented in Section 2.3. The result summary is followed by the retrospective discussion in Section 2.4 and a summary of the chapter in Section 2.5.

2.1.1 Background

The results of this chapter were obtained through a case study at the software development department described in Section 1.3.2. Since the department develops market-driven products where the pace and competition is intense, high delivery precision and short time-to-market is crucial for success. Therefore, it was of high importance to minimize the lead-time of the development process at the department. Further, since

the managers at the department believed that large gains could be obtained within the testing process; they requested further research within that area.

The purpose of the case study was to evaluate the development process used by the department and make a proposal for how the department could improve the test efficiency. In this context, improved test efficiency includes all aspects that affect the test lead-time, not only how efficient a certain test technique is at finding faults. That is, as described in Section 1.1, test efficiency does in this thesis foremost not consider how many faults a certain technique find divided by the required effort, but rather how a product reaches a sufficient quality level at the lowest cost. In order to achieve the specified purpose, the evaluation had the following objectives:

- Identify and evaluate "state of the art" techniques and processes
- Evaluate the development process at the department and identify improvements
- Propose how to apply the identified improvements at the department

2.2 Method

This section describes how the objectives presented in the previous section were achieved. The overall approach for how to conduct the evaluation comprised the following steps:

Study current research in software testing: Books and published articles were studied in order to get a general view on how software testing should be performed.

Practical evaluation of the test process: The most important part of the case study evaluation was to gather information about structure and practice of the test process used at the studied department. The primary purpose of the evaluation was to identify potential areas of improvement.

Make an improvement proposal based on the research results: From the case study evaluation and literature study, possible improvements were identified and analyzed. With support from gathered knowledge about the test process at the studied department, an implementation proposal for the feasible improvements was developed.

2.2.1 Data Collection

Selecting appropriate methods for data collection involved a few considerations. When conducting the case study, possible information sources were project statistics, interviews, questionnaires, and work observation. Since using many sources strengthens the validity of the results (Martella et al. 1999), (Robson 2002), all sources that were feasible to apply were selected. Of the possible information sources, work observation

was excluded since it would have been too time-consuming. The utilized information sources were applied as follows.

Interviews: To get an overall view of the case study setting and its processes, some informal interviews were conducted first, i.e. as 'informal conversational interviews' (Martella et al. 1999). The purposes of these interviews were to identify what project statistics that would be of interest and to identify questions to later seek answers to. After that, more structured interviews were conducted. The questions for these interviews were not developed according to a certain pattern. Instead, the interviews were adapted individually for each employee depending on the roles and areas of expertise they had, i.e. using the 'general interview guide approach' (Martella et al. 1999). Having loosely designed interviews allowed the employees to contribute with their thoughts without manipulation by the questioner. As a total, 15 people having various roles participated in these interviews.

Questionnaires: The purpose of handing out a questionnaire was to obtain quantitative results that could support the qualitative results. The questions were based on information obtained during the interviews. The main criteria considered when designing the questionnaire were that the answers should give numerical data that was important to have as support for the result analysis and it should only include questions that the employees had adequate knowledge for answering. The second criterion was ensured through pilot tests before sending out the questionnaire, i.e. two potential interviewees were handed preliminary versions of the questionnaire to determine if they understood the questions and thought that they would be able to answer them. In questions where the developers were supposed to weight the importance of different aspects against each other, a method called 'The 100-dollar test' was applied (Leffingwell and Widrig 2000). When using this method, the respondents are given 100 imaginary units to distribute between the selectable choices. This approach gives the respondents freedom not only to state that a choice is more important than another but also how much more important it is. All the 15 developers that were selected for this enquiry answered the questionnaire.

Project statistics: This evaluation part gathered data from a few finished and ongoing projects. Among the project documents, project evaluations that identified problems and possible improvements were considered especially useful for the qualitative study. When collecting data about efforts put on the test activities, it was discovered that the data in the time reporting system was unreliable. The primary reason for this was because the activity definitions were ambiguous. However, accurate lead-time data could at least be collected from various project documentation like for example progress reports. Fault reports were first also considered but turned out to be too time-consuming to include within the scope of the evaluation project.

2.3 Case Study Results

From the method described in the previous section, a set of assessment results was assembled and a number of improvements identified. These results are described as follows. The first section presents the techniques obtained from the literature study were considered interesting for the studied department. After that, the second section describes the key areas that were believed to be cost-effective to improve. Finally, the candidate improvements are listed sorted after whether they also were selected as concrete improvement proposals or not.

2.3.1 Literature Study

The literature review foremost included studies on state of the art in the area of software testing, and the aspects below were considered as potential contributors when identifying improvements at the studied department.

Orthogonal Defect Classification (ODC) is a technique that enables in-process feedback to developers from historical fault data. ODC is based upon the principle that different types of faults should be found in different test phases. ODC can detect problems when the distribution of a certain fault type in a certain phase is not normal (Chillarege et al. 1992).

Risk-based testing attempts to determine what parts of the system that pose the highest risks and then to put more efforts there so that the most harmful faults are exposed (McGregor and Sykes 2001). The idea with the technique is to focus testing and spend more time on critical functions. The technique calculates the risk exposure for each function by multiplying the probability of a fault to occur with the cost it would bring if not found until operation (Amland 2000).

Automated test tools of various kinds become increasingly important for making the test process more efficient. Scripts are a common way of automating tests, including not only the actual test execution but also the activities that are performed before and after, i.e. as further described in Section 1.1.2. Automated tools can be useful for most activities in the test process but they will only be beneficial when used in the right way (Fewster and Graham 1999).

Component testing has emerged due to an increased need for modularity of product architectures. That is, component based systems are increasingly used in larger software systems and therefore, new efficient test techniques for this area are required. Since components normally are object-oriented, approaches for testing object-oriented systems can be applied on components as well. Component testing is also comparable to traditional unit testing, but components are more complex, especially because they usually are not state-less (McGregor and Sykes 2001).

2.3.2 Case Study Assessment

The interviews, questionnaire, and project statistics together provided a rather clear view on the state of practice at the studied department. The most vital findings were summarized into the following four areas:

Many faults were more expensive to correct than necessary. As also confirmed in the literature study (Boehm 1983), (Shull et al. 2002), the case study evaluation results stated that faults are significantly more expensive to find and repair in later test phases. For example, both statistical data and questionnaire responses indicated the rework cost to be about seven times more expensive in function test compared to unit test. Additionally, indications of significantly higher fault cost rates in the maintenance phase were observed. Although the employees at the department were aware of this, more faults than necessary slipped through to later test phases. The findings below explain the reasons for that.

The amount of performed testing in earlier test phases was commonly insufficient because of...

...limited tool support for the developers and testers. The most frequently mentioned areas in the interviews regarded the test and debugging tools. The available tools were not considered to provide complete possibilities for testing all the behavior in the components, which made it hard to achieve a high code coverage. When in the questionnaire prioritizing causes of insufficient unit testing according to the 100-dollar method (see Section 2.2), the average score for this area was 33 dollars, i.e. the second highest rank.

...deadline pressure. Deadline pressure during development sometimes resulted in neglected testing and a lack of resources for improving the test environment. This area became important to consider when identifying reasons for incomplete unit testing. In the questionnaire, this area was given the highest rank, i.e. 45 dollars of the average score.

...limited knowledge on how to use the test tools. The developers stated that limited knowledge on how to use the existing tools decreased the amount of performed unit testing. This area was only given 14 dollars of the score in the questionnaire rank. However, the score still indicated some improvement potential in the area.

It should be noted that another area was also given a part of the scores in the questionnaire, i.e. that the reason for neglected unit testing would be because it was not required before delivery. However, since this area only was given eight dollars of the average score the weight was considered too low for being considered as a significant cause.

2.3.3 Identified Improvements

The qualitative interviews and post-mortem project evaluation reports identified several possible improvements. The list below specifies all the identified improvements that could aid in increased test efficiency.

Function specifications: The introduction of function specifications meant that the requirements were to be broken down into concrete specifications of how the requirements should be realized before starting detailed design of the product behavior. This should lead to less requirements related issues during testing. Further, it is cheaper to do it right the first time (Humphrey 2002).

Automated function test: A protocol-testing tool called TTCN3 was decided to be introduced in upcoming projects (ETSI 2001). This should lead to more efficient testing and also decrease the amount of regression faults found in later testing. The potential of automated testing was also acknowledged in the literature study.

Improvement of the product packaging process: To decrease the amount of manual work during product deliveries, implementations of more script support was requested. A decreased amount of human intervention would also decrease the risk for packaging related configuration faults after delivery.

Code inspections: Code inspections were in overall considered cost-effective to apply on the source code, and were suggested to be more extensively used. State of the art research also indicates potentials for improvements in this area (Aurum et al. 2002). However, as further discussed in Section 2.3.5, the certainty to obtain increased cost-efficiency from this improvement was not clear. Therefore, the given recommendation was to only apply inspections on critical code segments.

Specialist competence for inspections: This area is closely related to the previous one and the source of the improvement was that those that have significant competence on the specific product could identify significantly more faults in the code than others.

Improved version handling: Decreasing the amount of faults due to incorrect code versions would save time after internal deliveries. However, since these faults seemed to occur mostly due to human mistakes, specific improvement proposals were hard to identify.

Simulation tools: There was a request for allocation of more resources for development of simulation tools since they caused delays and faults during testing, i.e. because of a lack of time to have all the tool functionality ready and verified in time.

Risk-based testing: This area was considered promising since it could make testing more focused and thereby both save time and reduce the amount of customer faults (Amland 2000).

Improved unit test environment: This improvement should include improved unit test tools for testing the components developed at the department. Further, the unit test

process needed improvements that would ensure that a sufficient amount of unit tests would be performed before delivery to the function test phase.

Improved debugging environment: Debugging during bug fix was considered more time-consuming than necessary and the improvements suggested during the case study evaluation was to develop more support for trace and logging.

2.3.4 Improvement Selection

The previous section listed ten candidate improvements that were identified during the case study. However, several were for various reasons not feasible to make concrete improvement proposals for. This section describes the choice made for all the identified improvement candidates together with motivations for the choices.

Three of the candidate improvement already had concrete improvement proposals since they during the case study already were decided to be implemented:

- Function specifications
- Automated function test
- Improvement of the product packaging process

The next three candidate improvements listed below were considered cost-effective but did not result in any improvement proposals. The reasons for this was in the first two cases because it was primarily a matter of resource allocations in the projects and in the latter case because concrete improvement proposals could not be identified within the scope of the evaluation project:

- Code inspections
- Specialist competence for inspections
- Improved version handling

The next two candidate improvement were rejected because they were not considered cost-effective to introduce. In the first case because these tools were about to be replaced by a new third-party tool as described above (TTCN3). The second improvement was not suitable to implement at the same time as the test automation tools. That is, as stated earlier in this thesis, a company should select a few improvements and focus on those (Humphrey 2002):

- Simulation tools

- Risk-based testing

Finally, improvement proposals were developed for the two remaining identified improvements. In short, the improvement proposals for these improvements were as follows.

Improved unit test environment: This improvement proposal comprised a new tool for automated testing of the isolated components that were developed at the department. Further, the tool should be integrated with the test process in a way that would ensure that the developers would know how to use it and that the results of the test executions would be visible so that it would be possible to track that a sufficient amount of unit tests would be performed before delivery to the function test phase. Since this improvement proposal addressed all three causes of insufficient testing in earlier phases, it was considered as the most important improvement to implement.

Improved debugging environment: The improvement proposal for improved debug support included a suggestion for how to introduce a new third-party developed debugger. Further, it included a design suggestion for a run-time configurable debug component that could make it easier to trace and interpret the logs sent through the system.

2.3.5 Validity Threats to the Results

Since this case study was conducted as an isolated assessment where no improvements or experiments were made, the validity threat to the results to foremost consider was conclusion validity, i.e. the reliability of the results (Wohlin et al. 2003). First of all, subjectivism could occur for the investigators, the employees, and the researcher, whose opinions contribute to the results (Robson 2002). However, since the researcher was able to observe the projects closely, the ability to judge the reliability of the input increased. Additionally, the reliability was further ensured through triangulation of the results, i.e. several sources were used to obtain the results. Thus, it was possible to validate these sources against each other.

Regarding the questionnaire, the respondents might have interpreted the question on the efficiency of code inspections code differently. That is, some respondents might have thought that code inspections would increase the test efficiency in the current test process of the studied projects. However, with improvements that would give more efficient unit testing, the benefits might decrease when fewer faults that would have been found by inspections would slip through to later test phases. Further, since many of the respondents had limited experiences with code inspections, one could question their judgements.

When project staff estimate the amount of time they put on an activity they tend to estimate lower than the actual value (Nicholas 2001). A major reason for this is in our experience that people tend to forget some parts that are included in the activity. Therefore, the obtained values especially regarding average fault cost might be even higher than stated in the results. Nevertheless, this would in that case just strengthen the impact of those results even more, i.e. that many faults are more expensive to correct than needed.

2.4 A Retrospective View on the Case Study Evaluation

Almost three years have passed since the execution of the case study presented in this chapter. The purpose of this section it to provide an updated view on the results of that case study, i.e. if the results still can be considered valid and the status of the suggested improvements. The assessment results and the identified improvements are discussed in the corresponding subsections below. Also note that the literature study results presented in 2.3.1 also are considered in the subsection discussing the identified improvements.

2.4.1 Status of Assessment Issues

As stated in Section 2.3, some faults are still more expensive to correct than needed. However, as described in Chapter 5, improvements have been made regarding reducing the amount of faults in later phases. Further, as described in the next subsection, an increased amount of test automation has also decreased the average fault cost by decreasing the cost of re-testing bug fixes. Regarding the three main causes of insufficient unit testing, some improvements have been made:

- The tool support has as described in the next section been improved through a new test tool and an improved debug environment.
- Deadline-pressure still has a negative effect on early quality assurance and will most likely always have. However, as described in Chapter 4, introducing a new tool and applying it together with test-driven development, i.e. writing the tests before the code reduced the effect of the deadline-pressure. Chapter 4 and Chapter 5 further describe the implementation and results of it.
- When introducing the new unit test tool, training was given a high importance. The result of this has not been explicitly validated but since the tool usage is significantly higher now, the tool knowledge can at least be assumed to be better.

2.4.2 Improvement Status

Section 2.3.3 listed ten possible improvements of which two were rejected. Of the other eight improvements, seven have now at least been partially implemented. The current status of the improvements is as follows.

The most important improvement, i.e. '*Improved unit test environment'* was implemented as a part of the research presented in this thesis. The results of this improvement are described in Chapter 5. Further, '*Improved debugging environment'*, i.e. the other improvement with a concrete improvement proposal was implemented by employees at the department but not exactly as proposed in the improvement proposal. The results of this improvement have not been explicitly evaluated. Further, the introduction of '*Function specifications'* is at the department considered as a successful improvement. This is also the case for the introduction of '*Automated function test'* that despite upfront costs has started to pay off since the regression test time decreases significantly. Further, the '*The product packaging process'* is no longer considered an issue**.** Tool support for **'***Version handling'* has been improved but human mistakes still occur. **'***Code inspections'* are being used more and more but is not a clear exit criterion before delivery to the test department. The main reason for this is that it still is uncertain to what extent code inspections are cost-effective.

The identified improvement to allocate '*Specialist competence for inspections'* has not been implemented since such resource allocation does not work simply because these specialists are never available for such tasks due to a high workload. Finally, the two rejected improvements are still not implemented, i.e. in-house developed 'Simulation tools' are almost not used at all anymore and '*Risk-based testing'* is still not considered. However, the underlying objective of risk-based testing is implicitly covered in the new test strategy work presented in Chapter 6. Additionally, except for risk-based testing, the identified state of the art techniques from the literature review are central to the research presented in this thesis, i.e. techniques for automated testing of components and fault classification through Orthogonal Defect Classification (ODC).

2.5 Conclusions

This chapter presented a summary of results from a case study evaluation that was the starting point of the research presented in this thesis. The purpose of the case study was to determine how a software development department could shorten the test-lead time. The study evaluated the department's development process and made a proposal for how it could improve the test efficiency. The evaluation method consisted of three

steps:

- Study current research in software testing
- Perform a practical case study evaluation
- Make an improvement proposal based on the research results

Further, the data collected during these steps was obtained from interviews, various project statistics, and a questionnaire.

The main assessment result was that the cost of rework could have been reduced. The primary reason for this was because many faults that could have been found in earlier test phases instead slipped through to later stages of the projects. In a quantitative analysis, the three main causes of this were identified. In order of importance, the three causes were time-pressure, limited tool support, and insufficient knowledge on how to use the existing tools.

From the collected case study data, a number of improvements were suggested and concrete improvement proposals were made for two of them, i.e. improved unit test environment and improved debugger environment. Altogether, improvements in the test process that would result in earlier fault detection was considered to be the most important improvement area.

Finally, since the case study was completed almost three years ago, the last section provides a retrospect view on the current status of the assessment results and identified improvements. The analysis stated that the identified improvement areas have been addressed with positive results, both through implementations of suggested improvements and through further research results presented in this thesis.

Chapter 3

Phase-Oriented Process Assessment using Fault Analysis

Lars-Ola Damm, Lars Lundberg, and Claes Wohlin

Abstract

In market-driven development where time-to-market is of crucial importance, software development companies seek improvements that can decrease the lead-time and improve the delivery precision. One way to achieve this is by analyzing the test process since rework commonly accounts for more than half of the development time. A large reason for high rework costs is fault slippages from earlier phases where they are cheaper to find and remove. As input to improvements, this paper introduces a measure that can quantify this relationship. That is, a measure called faults-slip-through, which determines which faults that would have been more cost-effective to find in an earlier phase. The method presented in this paper also determines the excessive costs of the faults that slipped through phases, i.e. the improvement potential. The method was validated through practical application in two development projects at Ericsson AB. The results of the case study determined that the implementation phase had the largest improvement potential in the two studied projects since it caused a large faults-slip-through to later phases, i.e. 85 and 86 percent of the total improvement potential of each project.

3.1 Introduction

Reducing development costs and time-to-market while still maintaining a high level of product quality is essential for many modern software development organizations. These organizations seek specialized processes that could give them rapid improvements. However, they often overlook existing routinely collected data that can be used for process analysis (Cook et al. 1998). One such data source is fault reports since avoidable rework accounts for a significant percentage of the total development time, i.e. between 20-80 percent depending on the maturity of the development process (Shull et al. 2002). In fact, related research states that fault analysis is the most promising approach to software process improvement (Grady 1992).

A software development department at Ericsson AB develops component-based software for the mobile network. In order to stay competitive, they run a continuous process improvement program where they regularly need to decide where to focus the current improvement efforts. However, as considered a common reality in industry, the department is already aware of potential improvement areas; the challenge is to prioritize the areas to know where to focus the improvement work (Wohlwend and Rosenbaum 1993). Without such a decision support, it is common that improvements are not implemented because organizations find them difficult to prioritize (Wohlwend and Rosenbaum 1993). Further, if a suggested improvement can be supported with data, it becomes easier to convince people to make changes more quickly (Grady 1992). As stated above, fault statistics is one useful information source in software process improvement; therefore, the general research question of this paper is:

How can fault statistics be used for assessing a test process and quantifying the improvement potential of it in a software development organization?

Classification of faults from their causes, e.g. root cause analysis, is a commonly used approach for fault analysis (Leszak et al. 2000). Although root cause analysis can provide valuable information about what types of faults the process is not good at preventing/removing, the technique is cost intensive and therefore not easy to apply on larger populations of faults. Additionally, root cause analysis does not measure what the total cost of the slipped faults is. A test-oriented approach for fault analysis is fault trigger classification, which categorizes faults after which test activity that triggered them (Chillarege and Prasad 2002). This technique can be used for identifying how good a phase is at finding the faults it should find. However, as for root cause analysis, the technique cannot quantify the cost of faults that should have been found earlier.

To be able to address the stated research question, this paper instead introduces a method in which a central part is a ‘faults-slip-through’ measure. That is, the faults

are classified according to whether they slipped through the phase where they should have been found. This approach to fault classification has been used before (Basili and Green 1994), (Hevner 1997). However, the method presented in this paper has two main differences. First, faults-slip-through focuses on when it is cost-effective to find each fault, it does not consider fault insertion phase. Second, the presented method also calculates the improvement potential by relating the result of the fault classification with the average cost of finding and repairing the faults. That is, the method measures the improvement potential by multiplying the faults-slip-through distribution with the average benefit of finding a fault earlier.

In order to validate the applicability of the method, the paper also provides an empirical case study where the method is applied on the faults reported in two finished development projects at Ericsson AB. The paper is outlined as follows. Section 3.2 presents some related work and Section 3.3 describes the proposed method for how to determine the improvement potential of an organization. Section 3.4 demonstrates the applicability of the method through an empirical case study. Section 3.5 discusses the validity and implications of the results and Section 3.6 concludes the work.

3.2 Related Work

The Capability Maturity Model (CMM) is one of the most widely used models for software process improvement (Paulk et al. 1995). The essence of this model and its tailored variant SW-CMM (SW-CMM) is for organizations to strive for achieving higher maturity levels and thereby become better at software development. However, such blueprint models have been criticized because they are focused on making an organization more structured and work according to what is defined to be state of the art in the model. That is, although state of the art models can provide some guidance to areas of improvement, there are no generally applicable solutions (Glass 2004), (Mathiassen et al. 2002).

The opposite of applying a model-based approach is the bottom-up approach where improvements are identified and implemented locally in a problem-based fashion (Jakobsen 1998). A typical bottom-up approach is the Quality Improvement Paradigm (QIP), where a six-step improvement cycle guides an organization through continuous improvements based on QIP (Basili and Green 1994). From defined problem based improvements, QIP sets measurable goals to follow up after the implementation. However, QIP is more of a generic framework for which steps to include in an improvement cycle, it does not state exactly how to perform them. That is, in practice, the method in this paper could instead be included as a part of QIP, i.e. by including faults-slip-through as an important measure to follow up with respect to fault cost reduction.

Within the area of fault-oriented measures, a widely spread fault measurement approach is ‘Six sigma‘, which is centered on a measure that determines the number of faults in relation to product size (Biehl 2004). Further, as earlier mentioned, the measure applied in this paper is similar to measures used in related work, e.g. phase containment metrics where faults should be found in the same phase as they were introduced (Hevner 1997), and goodness measures where faults should be found in the earliest possible phase (Berling and Thelin 2003). In contrast to the faults-slip-through measure, these measures are strongly related to the notion of fault latency, i.e. for how long time does a fault remain in a product. The implication of this is that since most faults are inserted during analysis, design and coding, the measures primarily provide feedback on earlier phases instead of the test process. Therefore, they are not suitable for improvements aimed at the test process.

Finally, related work on calculating the improvement potential from faults has been done before, i.e. by calculating the time needed in each phase when faults were found when supposed to in comparison to when they were not (Tanaka et al. 1995). Although the results of using such an approach are useful for estimating the effect of implementing a certain improvement, they require measurements on the additional effort required for removing the faults earlier. Such measurements require decisions on what improvements to make and estimates of what they cost and therefore they cannot be used as input when deciding in which phases to focus the improvements and what the real potential is.

3.3 Method

3.3.1 Estimation of Improvement Potential

The purpose of this paper is to demonstrate how to determine the improvement potential of a development process from historical fault data. This section describes the selected method for how to achieve this through the following three steps:

(1) Determine which faults that could have been avoided or at least found earlier

(2) Determine the average cost of finding faults in different phases.

(3) Determine the improvement potential from the results in (1) and (2).

In this context, a fault is defined as an anomaly that causes a failure (IEEE 1988). The following three sub-sections describe how to perform each of the three steps.

Faults-slip-through measurement

When using fault data as basis for determining the improvement potential of an organization's development process, the essential analysis to perform is whether the faults could have been avoided or at least have been found earlier. As previously mentioned, the introduced measure for determining this is called 'faults-slip-through', i.e. whether a fault slipped through the phase where it should have been found. The definition of it is similar to measures used in related studies, e.g. phase containment metrics where faults should be found in the same phase as they were introduced (Hevner 1997), and goodness measures where faults should be found in the earliest possible phase (Basili and Green 1994). The main difference between the faults-slip-through measure and the other measures is when a fault is introduced in a certain phase but it is not efficient to find in the same phase, e.g. a certain test technique might be required to simulate the behaviour of the function. Then it is not a faults-slip-through. Figure 1.3 in Chapter 1 further illustrates this difference.

A consequence of how faults-slip-through is measured is that a definition must be created to support the measure, i.e. a definition that specifies which faults that should be found in which phase. To be able to specify this, the organization must first determine what should be tested in which phase. Therefore, this can be seen as test strategy work. Thus, experienced developers, testers and managers should be involved in the creation of the definition. The results of the case study in Section 4 further exemplify how to create such a definition.

Table 3.1 provides a fictitious example of faults-slip-through between arbitrarily chosen development phases. The columns represent in which phase the faults were found (PF) and the rows represent where the faults should have been found (Phase Belonging, PB). For example, 25 of the faults that were found in Function Test should have been found during implementation (e.g. through inspections or unit tests). Further, the rightmost column summarizes the amount of faults that belonged to each phase whereas the bottom row summarizes the amount of faults that were found in each phase. For example, 49 faults belonged to the implementation phase whereas most of the faults were found in Function Test (50).

Average Fault Cost

When having all the faults categorized according to the faults-slip-through measure, the next step is to estimate the cost of finding faults in different phases. Several studies have shown that the cost of finding and fixing faults increases more and more the longer they remain in a product (Boehm 1983), (Shull et al. 2002). However, the cost-increase varies significantly depending on the maturity of the development process and

Table 3.1: Fictitious example of faults-slip-through data

PF: / PB:	Design	Impl.	Function Test	System Test	Operation	Total belonging/phase
Design	1	1	10	5	1	**18**
Impl.		4	25	18	2	**49**
Function Test			15	5	4	**24**
System Test				13	2	**15**
Operation					0	**0**
Tot. found/phase	**1**	**5**	**50**	**41**	**9**	**106**

on whether the faults are severe or not (Shull et al. 2002). Therefore, the average fault cost in different phases needs to be determined explicitly in the environment where the improvement potential is to be determined (see a fictitious example in Table 3.2). This measure could either be obtained through the time reporting system or from expert judgments, e.g. a questionnaire where the developers/testers that were involved in the bug-fix give an estimate of the average cost.

Table 3.2: Fictitious example of average fault cost/phase found

	Design	Implementation	Function Test	System Test	Operation
Average fault cost	1	2	10	25	50

Expert judgments are a fast and easy way to obtain the measures; however, in the long-term, fully accurate measures can only be obtained by having the cost of every fault stored with the fault report when repairing it. That is, when the actual cost is stored with each fault report, the average cost can be measured instead of just being subjectively estimated. Further, when obtaining these measures, it is important not just to include the cost of repairing the fault but also fault reporting and re-testing after the fault is corrected.

Improvement Potential

The third step (e.g. the improvement potential) is determined by calculating the difference between the cost of faults in relation to what the fault cost would have been if none of them would have had slipped through the phase where they were supposed to be found. Figure 3.1 provides the formulas for making such a calculation and as presented in the table in the figure, the improvement potential can be calculated in a

two-dimensional matrix. The equation in the figure provides the actual formula for calculating the improvement potential for each cell (IP_{xx}). PF_{xtotal} and PBx_{total} are calculated by summarizing the corresponding row/column. As illustrated rightmost in the figure, the average fault cost (as discussed in the previous paragraph), need to be determined for each phase before using it in the formula (IP_{xx}).

In order to demonstrate how to use and interpret the matrix, Table 3.3 provides an example calculation by applying the formulas in Figure 3.1 on the fictitious values in Table 3.1 and Table 3.2. In Table 3.3, the most interesting cells are those in the rightmost column that summarizes the total cost of faults in relation to fault belonging and the bottom row that summarizes the total unnecessary cost of faults in relation to phase found. For example, the largest improvement potential is in the implementation phase, i.e. the phase triggered 710 hours of unnecessary costs in later phases due to a large faults-slip-through from it. Note that taking an action that removes the faults-slip-through from the implementation phase to later phases will increase the fault cost of the implementation phase, i.e. up to 49 hours (1 hour/fault times 49 faults that according to Table 3.2 belonged to the implementation phase). Further, System Test is the phase that suffered from the largest excessive costs due to faults slipped through (609 hours). However, when interpreting such excessive costs, one must be aware of that some sort of investment is required in order to get rid of them, e.g. by adding code inspections. Thus, the potential gain is probably not as large as 609 hours. Therefore, the primary usage of the values is to serve as input to an expected Return On Investment (ROI) calculation when prioritizing possible improvement actions.

	PF_1	PF_2	PB_{total}	AvFC
PB_1	IP_{11}	IP_{12}	$PB_{1\ total}$	PB_1 AvFC
PB_2	IP_{21}	IP_{22}	$PB_{2\ total}$	PB_2 AvFC
PF_{total}	$PF_{1\ total}$	$PF_{2\ total}$	$(PB/PF)_{total}$	

PF = Phase found, PB =Phase belonging, AvFC=Average Fault Cost
IP = Improvement potential

IP_{xx} = (Nr faults bel. (PBx) * PB_1 AvFC) – (Nr faults bel. (PBx) * PBxAvFC)

Figure 3.1: Matrix formula for calculation of improvement potential

When measured in percent, the improvement potential for a certain phase equals the improvement potential in hours divided with the total improvement potential (e.g. in the example provided in Table 3.3, the improvement potential of System Test = 609/1255=49%). In the case study reported in the next section, the measurements are provided in percent (due to confidentiality reasons).

Table 3.3: Example of calculation of improvement potential (hours)

PF: / PB	Design	Impl.	Function Test	System Test	Operation	Total PB/phase
Design	1*1-1*1 = 0	1*2-1*1 = 1h	10*10-10*1 = 90h	5*25-5*1 = 120	1*50-1*1 = 49	**260h**
Impl.		4*2-4*2 = 0	25*10-25*2 = 200h	18*25-18*2 = 414h	2*50-2*2 = 96h	**710h**
Function Test			15*10-15*10 = 0	5*25-5*10 = 75h	4*50-4*10 = 160h	**235h**
System Test				13*25-13*25 = 0	2*50-2*25 = 50h	**50h**
Operation					0	**0h**
Total potential/ PF		**1h**	**290h**	**609h**	**355h**	**1255h**

3.4 Results from Applying the Method

This section describes the case study application of the previously defined described method. First, Section 3.4.1 provides an overview of the case study environment. After that, sections 3.4.2-3.4.4 describes the result of applying the three steps of the method.

3.4.1 Case Study Setting

The applicability of the described method was evaluated by using it on the faults reported in two projects at a department at Ericsson AB. The projects developed functionality to be included in new releases of two different products. Hence, previous versions of the products were already in full operation at customer sites. Further, the projects used the same processes and tools and the products developed in the projects were developed on the same platform (i.e. the platform provides a component-based architecture and a number of platform components that are used by both products). The products were developed mainly in C++ except for a Java-based graphical user interface that constitutes minor parts of each product. Apart from the platform components, each product consists of about 10-30 components and each component consists of about 5-30 classes. The reason for studying more than one project was to be able to strengthen the validity of the results, i.e. two projects that were developed in the same environment and according to the same development process should provide similar results (except for known events in the projects that affected the results). Further, two projects were chosen since the selected projects were the only recently finished projects

and because earlier finished projects were not developed according to the same process. Thereby, it was the two selected projects that could be considered as representative for the organization.

The reported faults originated from the test phases performed by the test unit at the department, i.e. faults found earlier were not reported in a written form that could be post-analyzed. Further, during the analysis, some faults were excluded either because they were rejected or because they did not affect the operability of the products, e.g. opinion about function, not reproducible faults, and documentation faults. Finally, requirements faults were not reported in the fault reporting system. Instead, they were handled separately as change requests.

3.4.2 Faults-Slip-Through

Figure 3.2 and Figure 3.3 present the average percent faults-slip-through in relation to percent faults found and development phase from two finished projects at the department. The faults-slip-through measure was not introduced until after the project completions, and hence all the fault reports in the projects studied needed to be classified according to the method described in Section 3.3.1 in retrospect. The time required for performing the classification was on average two minutes/fault. Actually, several faults could be classified a lot faster but some of them took a significantly longer time since these fault reports lacked information about the causes of the faults. In those cases, the person that repaired the fault needed to be consulted about the cause. Additionally, in order to obtain a consensus on what faults should be considered as faults-slip-through and not, a workshop with key representatives from different areas at the department was held. The output from the workshop was a definition of which faults that should belong to which phase. Appendix C presents the obtained definition for each phase. However, note that the two last phases below (FiT+6, FiT_7-12) were not included in the definition since at that stage all faults were considered as faults-slip-through. When assigning faults to different phases, the possible phases to select among were the following:

Implementation (Imp): Faults found when implementing the components, e.g. coding faults found during unit tests.

Integration Test (IT): Faults found during primary component integration tests, e.g. installation faults and basic component interaction faults.

Function Test (FT): Faults found when testing the features of the system.

System Test (ST): Includes integration with external systems and testing of non-functional requirements.

Field Test + 6 months (FiT+6): During this period, the product is tested in a real environment (e.g. installed into a mobile network), either at an internal test site or

together with a customer. During the first six months, most issues should be resolved and the product then becomes accepted for full operation.
Field Test 7-12 months (FiT_7-12): Same as FiT+6; however, after 6 months of field tests, live usage of the product has normally begun.

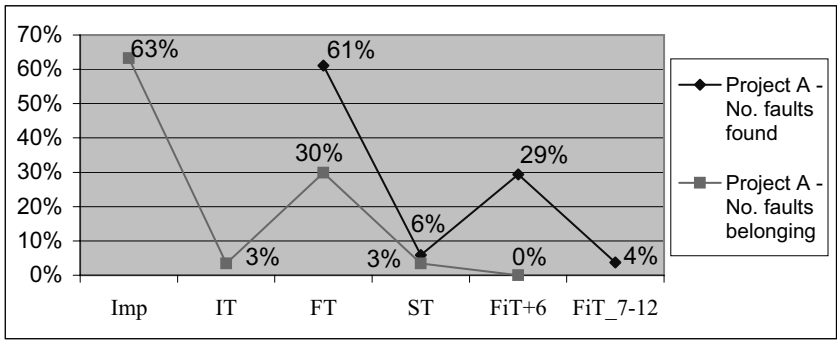


Figure 3.2: Percent faults-slip-through in relation to percent faults found and development phase (Project A)

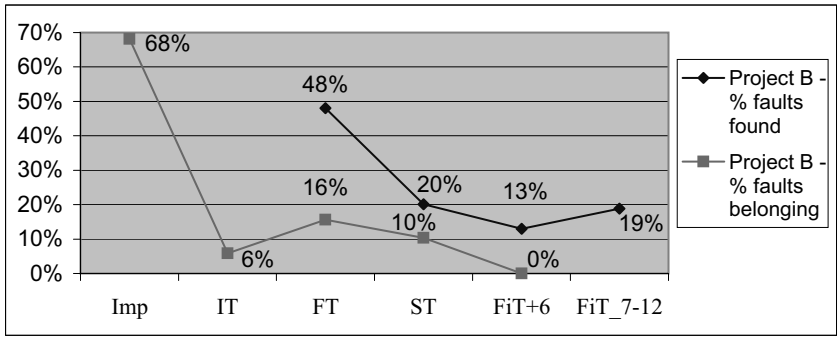


Figure 3.3: Percent faults-slip-through in relation to percent faults found and development phase (Project B)

Regarding the possible phases to select between, it could have been possible to include earlier phases such as requirements analysis and system design. However, since the studied department wanted to focus on feedback on the test phases when using this measure, earlier phases were excluded.

As can be seen in the figures, several faults belonged to the implementation phase in the two projects, i.e. 63 and 68 percent respectively. Further, in project A (Figure 3.2), many faults were found in FiT+6 (29%). The primary reason for this was that the field tests started before ST was completed, i.e. the phases were overlapping which resulted in that ST continued to find faults during FiT+6. These ST faults could for practical reasons only be classified as FiT+6 faults.

The figures illustrate the faults-slip-through distributions of the projects in a good way. However, sometimes it is more feasible to present the measure as the total amount of faults-slip-through to a certain phase, e.g. at the studied department the measure was to be used as goal values in balanced scorecards (Kaplan and Norton 1996). That is, in this case only a few key goal measures were requested and therefore, a multi-valued graph was not feasible. Table 3.4 describes how this was applied at Ericsson AB. For example, in the table, 69, and 80 percent faults-slip-through to FT was calculated as a sum of all faults that should have been found in earlier previous phases divided with the number of faults found in FT.

Table 3.4: Percent Faults-slip-through (FST) to Function Test and System Test

	Project A	**Project B**
FST to Function Test	69%	80%
FST to System Test	77%	77%

3.4.3 Average Fault Cost

When estimating the average fault cost for different phases at the department, expert judgments were used since neither was the fault cost reported directly into the fault reporting system nor was the time reporting system feasible to use for the task. In practice, this means that the persons that were knowledgeable in each area estimated the average fault cost. Table 3.5 presents the result of the estimations. For example, a fault costs 13 times more in System Test (ST) than in Implementation (Imp). In the table, the cost estimates only include the time required for reporting, fixing and re-testing each fault, which means that there might have been additional costs such as the cost of performing extra bug-fix deliveries to the test department. Such a cost is hard to account for since the amount of deliveries required is not directly proportional to the amount of faults, i.e. it depends on the fault distributions over time and the nature of the faults. The reason why FiT_7-12 was estimated to have the same cost as FiT+6 was because the system was still expected to be in field tests although live usage in reality actually might already have started. Further, during the first 12 months after the

field tests have started, few systems have been installed although the system becomes available for live usage already during this period. That is, the fault cost rises when more installed systems need to be patched, but, in reality, this does not take any effect until after FiT_7-12.

Table 3.5: Average fault-cost/phase at the department (in relative terms)

Phase found	**Imp**	**IT**	**FT**	**ST**	**FiT+6**	**FiT_7-12**
Average cost/fault	1	2.5	8.2	13	17	17

3.4.4 Improvement Potential

Table 3.6 and Table 3.7 present the improvement potential of the two studied projects from the fault statistics provided in Sections 3.4.2 and 3.4.3, calculated according to the method provided in Section 3.3.1. As can be seen in both tables, faults-slip-through from Implementation comprised a significant proportion of the improvement potential (85, 86%); therefore, this is foremost where the department should focus their improvement efforts. Further, in project B, all the test phases had a significant improvement potential, e.g. FT could be performed at a 32 percent lower cost by avoiding the faults-slip-through to it. On the contrary, project A had more diverse fault distributions regarding phase found. The reason for this is mainly due to overlapping test phases (further discussed in Section 3.4.2). Finally, it should also be noted that the total improvement potential in relation to fault origin phase (rightmost columns) were similar for both projects, which strengthens the assumption that the improvement potential is foremost process related, i.e. the faults-slip-through did not occur due to certain product problems or accidental events in the projects.

Table 3.6: Percent improvement potential (Project A)

Phase found **Phase belonging**	**FT**	**ST**	**FiT+6**	**FiT_7-12**	**Total potential /origin phase**
Imp	37	5.6	37	5.4	**85**
IT	2.2	0.0	0.5	0.5	**3.2**
FT	0.0	0.7	10.1	0.6	**11**
ST	0.0	0.0	0.8	0.1	**0.9**
Total potential/test phase	**39**	**6.3**	**48**	**6.6**	**100**

Table 3.7: Percent improvement potential (Project B)

Phase found / Phase belonging	FT	ST	FiT+6	FiT_7-12	Total potential /origin phase
Imp	30	16	12	28	**86**
IT	1.7	2.5	2.2	0.0	**6.4**
FT	0.0	1.6	1.3	2.0	**4.9**
ST	0.0	0.0	1.5	0.8	**2.3**
Total potential/test phase	**32**	**20**	**17**	**30**	**100**

3.5 Discussion

This section discusses the proposed method and the results of applying it. First, Section 3.5.1 presents a few lessons learned from defining and applying the central part of the method, i.e. the faults-slip-through measure. After that, Section 3.5.2 describes observed implications of the results. Finally, Section 3.5.3 discusses the validity threats to the results.

3.5.1 Faults-slip-through: Lessons learned

Applying faults-slip-through in an industrial environment gave several experiences regarding what works and not. First of all, creating the definition resulted in a few lessons learned:

- Start with defining the test strategy if it is not already clearly specified. That is, there is a direct mapping between what types of tests a phase should cover and which faults should be found when executing those tests.
- Specify the definition after the goal test strategy to reach within a few years time, not the current situation. This is very important to make the measure possible to improve against and to keep the definition stable. If the definition is changed between each project, the projects are not possible to compare against each other.
- Create the definition iteratively, i.e. develop a suggestion, test it on a set of faults, and then refine it iteratively until satisfactory. The reason for this is because it is during practical usage possible flaws are found, especially fault types that the involved people forgot to include in the definition.

When the definition was created, the measure was also added to the fault reporting process, i.e. to make sure that it would be reported when correcting the faults so that

a post-mortem analysis would not be needed after the next project. Experiences from doing this gave the following major lessons learned:

- The developers fixing the fault should specify the faults-slip-through measure in the fault reports. Testers can be good to use as support in some situations but in our experience, it is the developers that knows best in most cases. However, the later test phase a fault is found in, the more influence should the testers have since they know that area better. Further, the testers are always good to have as a point of validation to make sure that the value was reported correctly.
- Make sure to educate the developers and testers properly so that they have a common view on how to use it. Inevitably, there is a degree of subjectiveness in the definition but creating a common mindset can minimize this. Additionally, the first time the measure is applied in a project, someone that took part in creating the definition should validate the reported values to verify that everyone understood the measure correctly.
- Make sure that the values are easy to follow up, i.e. automate the data analysis as much as possible, e.g. generate the faults-slip-through results on a webpage or make the data easy to generate into a spreadsheet.

3.5.2 Implications of the Results

The primary implication of the results is that they provide important input when determining the Return On Investment (ROI) of suggested process improvements. That is, since software process improvement is about reducing costs, the expected ROI needs to be known; otherwise, managers might not want to take the risk to allocate resources for handling upfront costs that normally follow with improvements. Additionally, the results can be used for making developers understand why they need to change their ways of working, i.e. a quantified improvement potential motivates the developers to cooperate (Tanaka et al. 1995).

Improvement actions regarding the issues resulting in the largest costs were implemented in subsequent projects, e.g. more quality assurance in earlier phases. Besides shortening the verification lead-time, the expected result of decreased faults-slip-through percentages was to improve the delivery precision since the software process becomes more reliable when many faults are removed in earlier phases (Tanaka et al. 1995). The projects using the method will be studied as they progress.

An unanticipated implication of introducing the faults-slip-through measure was that it became a facilitator for discussing and agreeing on what to test when, e.g. improvement of test strategies. This implies that the measure could serve as a driver for

test process improvement. Finally, the definition turned out to be a good support for driving test process alignment work, i.e. by making different projects and products adhere to the same faults-slip-through definition. That is, it provides a way to specify how to work and then follow up if this way of working is achieved.

3.5.3 Validity Threats to the Results

When conducting an empirical industry study, the environment cannot be controlled to the same extent as in isolated research experiments. In order to be able to make a correct interpretation of the results presented in Section 4, one should be aware of threats to the validity of them. As presented below, the main validity threats to this case study concern conclusion, internal, and external validity (Wohlin et al. 2003). Construct validity is not relevant in this context since the case study was conducted in an industrial setting.

Conclusion validity concerns whether it is possible to draw correct conclusions from the results, e.g. reliability of the results (Wohlin et al. 2003). The threats to conclusion validity are as follows. First, in order to be able to draw conclusions from the results, the department must have a common view on which phase each fault should belong to. That is, managers and developers should together agree on which fault types that should be considered as faults-slip-through and not. At the studied department, this was managed by having workshops where checklists for how to estimate fault-slip-through were developed. However, continuous improvements and training are required in the first projects in order to ensure that everyone have the same view on how to make the estimations. Further, regarding the average fault cost for different phases, the result was obtained through expert judgments and therefore, the estimations might not exactly reflect the reality. However, this was minimized by asking as many 'experts' as possible, i.e. although there might be deviations, the results were good enough to measure the improvement potential from and hence use as basis for decisions. However, in the future, direct fault cost measures should be included in the fault reports so that this uncertainty is removed. Finally, since the improvement potential is calculated from the faults-slip-through measure and the average fault cost measure, the accuracy of the improvement potential is only dependent on the accuracy of the other measures.

Internal validity concerns how well the study design allows the researchers to draw conclusions from causes and effects, i.e. causality. For example, there might be factors that affect the dependent variables (e.g. fault distributions) without the researchers knowing about it (Wohlin et al. 2003). In the case study presented in Section 4, all faults were post-classified by one researcher, which thereby minimized the risk for biased or inconsistent classifications. Another threat to internal validity is whether certain events that occurred during the studied projects affected the fault distribution,

i.e. events that the researchers were not aware of. This was managed through workshops with project participants where possible threats to the validity of the results were put forward. Additionally, since two projects were measured, the likelihood of special events that affected the results without being noticed decreased. That is, the obtained improvement potentials of the two studied projects were very similar (85 and 86%), which indicates that the causes of the results were directly related to the projects' common denominator, i.e. that they used the same tests process.

External validity concerns whether the results are generalizable or not (Wohlin et al. 2003). In the performed case study, the results are not fully generalizable since they are dependent on the studied department having certain products, processes, and tools. However, since two projects were studied and gave similar fault distributions, the results are at least generalizable within the context of the department. Further, the results on average fault costs in different phases (see Section 3.4.4) acknowledge previous claims in that faults are significantly more expensive to find in later phases (Boehm 1983), (Shull et al. 2002). Nevertheless, in this paper, the main concern regarding external validity is whether the method used for obtaining the results is generalizable or not. Since the method contains no context dependant information, there are no indications in that there should be any problems in applying the method in other contexts. Thus, the method can be replicated in other environments.

3.6 Conclusions and Further Work

This paper presents and validates a method for measuring the efficiency of the software test process. The main objective of the paper was to answer the following research question:

How can fault statistics be used for assessing a test process and quantifying the improvement potential of it in a software development organization?

The method developed for answering the research question determines the improvement potential of a software development organization through the following three steps:

(1) Determine which faults that could have been avoided or at least found earlier.

(2) Determine the average cost of faults found in different phases.

(3) Determine the improvement potential from the measures in (1) and (2), i.e. measure the cost of not finding the faults as early as possible.

The practical applicability of the method was determined by applying it on two industrial software development projects. In the studied projects, potential improvements were foremost identified in the implementation phase, e.g. the implementation phase

inserted, or did not capture faults present at least, too many faults that slipped through to later phases. For example, in the two studied projects, the Function Test phase could be improved by up to 39 and 32 percent respectively by decreasing the amount of faults that slipped through to it. Further, the implementation phase caused the largest faults-slip-through to later phases and thereby had the largest improvement potential, i.e. 85 and 86 percent in the two studied projects.

The measures obtained in this report provide a solid basis for where to focus improvement efforts. However, in further work, the method could be complemented with investigations on causes of why faults slipped though the phase where they should have been found. For example, the distribution of faults-slip-through should be related to the underlying test activities in order to be able to focus more efforts on specific test activities that had a high faults-slip-through.

Chapter 4

A Framework for Test Automation and Test-Driven Development

Lars-Ola Damm, Lars Lundberg, and David Olsson

Abstract

This paper identifies and presents an approach to software component-level testing that in a cost effective way can move defect detection earlier in the development process. A department at Ericsson AB introduced a test automation tool for component-level testing in two projects together with the concept test-driven development (TDD), a practice where the test code is written before the product code. The implemented approach differs from how TDD is used in Extreme Programming (XP) in that the tests are written for components exchanging XMLs instead of writing tests for every method in every class. This paper describes the implemented test automation tool, how test-driven development was implemented with the tool, and experiences from the implementation. Preliminary results indicate that the concept decreases the development lead-time significantly.

4.1 Introduction

Studies indicate that testing accounts for at least 50% of the total development time (Hambling 1993), (Harrold 2000). One reason for this is that the verification activities late in development projects tend to be loaded with defects that could have been prevented or at least removed earlier (when they are cheaper to find and remove (Boehm 1983), (Hambling 1993), (Mosley and Posey 2002)). When many defects remain to be found late in a project, schedules are delayed and the verification lead-time increases (Humphrey 2002).

A software development department at Ericsson AB (from now on referred to as the department) develops component-based software for the mobile network. The department wanted to decrease the verification lead-time and avoid the risk for delayed deliveries by introducing a new tool and process for automated testing on a component level. To achieve this, they needed to determine what was required of the tool, process and organization.

Thus, the paper has four main questions to answer:

1. What characteristics should a test automation tool for component level testing have?
2. What is an appropriate process supporting the use of such a test automation tool?
3. What aspects need to be considered when introducing a new process and tool for component level testing?
4. What is the expected lead-time difference for the projects that introduce such a tool and process?

The method used for answering the first two questions was a combination of analysis of lessons learned from previous improvement attempts together with the results of a case study evaluation described in Chapter 2. The evaluation included qualitative and quantitative enquires, analysis of project statistics, and a literature study. Answers to the other two questions were captured from qualitative and quantitative interviews with users of the introduced concept.

The paper is outlined as follows. Section 4.2 gives a background including the choice of process for the concept, i.e. test-driven development. Section 4.3 presents the actual implementation together with some lessons learned and expected lead-time gains. Section 4.4 maps the new test automation tool against related techniques for test automation and then also describes what is required of an organization before being able to implement such a concept. Section 4.5 summarizes the work through some conclusions.

4.2 Background

The approach used before the introduction of this new concept comprised a test tool (called DailyTest) that could test isolated components before delivery to the function test department. In this context, a component is an executable asset that communicates with other components through common interfaces. Further, a component contains about 5-30 classes and the products that the department develops consist of about 10-30 components each.

DailyTest could execute scripts consisting of simple commands that for example loads a component and then sends a sequence of requests on it. The reasons why the tool did not work satisfactory were that it was limited (few commands and no looping) and more importantly, it was not properly integrated with the development process. When the department had realized this, the earlier mentioned case study described in Chapter 2 evaluated the test process with the purpose to identify tool and process changes that would increase the test efficiency. The evaluation showed that the cost of finding and fixing defects increases significantly the later in the development process they are found. This is also considered a common fact in software development (Boehm 1983), (Hambling 1993), (Mosley and Posey 2002). Since the case study also discovered that the developers normally put little effort on testing isolated components, the case study suggested that this is where to focus the improvement efforts.

Testing of isolated software components is the first test level in the department's quality assurance strategy, Basic Test in Ericsson terminology. As in ordinary unit testing, the purpose of Basic Test is to verify the design specification. However, as opposed to unit testing, Basic Test is not a white-box test technique since the components' interfaces hide the internal component design. Nevertheless, since it is the product developers that perform Basic Test on the components, they still know the internal design structure. The reason why the developers start doing testing on a component level is because it is not cost-effective for them to test every class/method. The reason for this is that the department's products have a higher testability on the component level, i.e. the components are independent executables and the XML requests that the components send between each other are easier to verify (testability is further discussed in Section 4.4.2).

After determining that the improvement efforts should focus on the Basic Test level, the case study determined that the main reasons for why the developers at the department did not Basic Test all functionality was because of insufficient test tools (e.g. DailyTest) and process deviations when the deadline pressure was high due to delayed schedules. When the development activities were delayed, the projects tended to deliver the code untested hoping that it in a miraculous way would work anyway. Likewise, this phenomenon seems far from uncommon in the software industry (Ince

1993), (Mosley and Posey 2002).

From the experiences and findings discussed above, the case study suggested a new tool for how to automatically Basic Test the products at the department, which they thereafter implemented and introduced in two upcoming projects.

4.2.1 Test-Driven Development

To make sure that the new Basic Test tool would not become a shelfware, the department put considerable efforts in integrating the tool with the development process and they chose between keeping their previous standard process or to introduce the new concept Test-Driven Development (TDD) (Beck 2003).

The main difference between TDD and a typical test process is that in TDD, the developers write the tests before the code. A result of this is that the test cases drive the design of the product since it is the test cases that decide what is required of each unit (Beck 2001), (Beck 2003). Therefore, TDD is not really a test technique (Beck 2001), (Cockburn 2002); it should preferably be considered a design technique. Furthermore, TDD simplifies the design and makes sure that the implementation scope is explicit, i.e. it removes the desire for gold plating (Beck 2001). Nevertheless, the most obvious advantage of TDD is the same as for test automation in general, i.e. the possibility to do continuous quality assurance of the code (including regression testing) (Fewster and Graham 1999). This gives both instant feedbacks to the developers about the state of their code and most likely, a significantly lower percentage of defects left to be found in later testing and at customer sites (Maximilien and Williams 2003). Further, with early quality assurance, a common problem with test automation is avoided; that is, when an organization introduces automated testing late in the development cycle, it becomes a catch for all defects just before delivery to the customer. The corrections of found defects lead to a spiral of testing and re-testing which delays the delivery of the product (Kehlenbeck 1997).

The most negative aspect concerning TDD is that in worst case, the test cases duplicate the amount of code to write and maintain. However, this is the same problem as for all kinds of test automation (Hayes 1995), and to what extent the amount of code increases depends on the granularity of the test cases and what module level the test cases encapsulates, e.g. class level or component level. Thereby, since the department would use TDD on a component level, they would decrease the amount of test case code to write and maintain. Additionally, in comparison to classes/methods, it is easier to automate uniform interfaces that use XML as data format and that are more robust to changes.

Nevertheless, the department foremost chose to use TDD because it can eliminate the previously mentioned risk for improperly conducted Basic Test. When the test

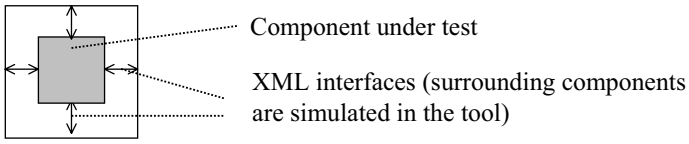


Figure 4.1: Basic Test tool - attachment to components

cases are developed before the code, it is consequently impossible to deliver the code without developing the test cases. Meanwhile, there is no reason not to Basic Test the product when the executable test cases already are developed. To summarize, the primary purpose of the new concept was to increase the amount of tested code in Basic Test.

4.3 Description of the Basic Test Concept

After providing some background information, sections 4.3.1 and 4.3.2 give a technical tool description and Section 4.3.3 describes how the tool was integrated with the development process at the department. After that, Section 4.3.4 lists some observations and lessons learned and finally, Section 4.3.5 presents the expected lead-time gains from introducing the concept.

4.3.1 Choice of Tool and Language

Since the purpose of Basic Test is to test the components in isolation, the Basic Test tool needed to be attached to the components' interfaces, i.e. simulating the surrounding components. This attachment is demonstrated in Figure 4.1.

DailyTest, the previous Basic Test tool the department used, attached to the components in a similar way as in Figure 4.1. The reason why it was not preferable to enhance DailyTest instead of developing a new tool was that to make DailyTest as powerful as needed, it would almost become a program language. Naturally, it is not beneficial to do that when there already exist several powerful standard languages such as C++ and Java.

After choosing not to enhance the existing test tool, the department needed to decide whether to develop a new tool using a standard language or buying a commercial tool. First of all, the TDD tools normally used in Extreme Programming (e.g. cppUnit) (Beck 2003) were not appropriate in this context since they operate on a class/method level; that is, they do not support component level interfaces (e.g. XML communica-

tion). Further, since Basic Test requires tight integration with the product architecture, the test tools that commercial vendors develop are hard to use for this purpose since they require some kind of interface that can connect to the components to test. Such an adaptation would involve additional costs and more importantly, such tools would put unwanted constraints on how to develop, execute, and monitor the tests. This combined with the fact that it was relatively cheap to develop an in-house tool when the product architecture had such high testability (see Section 4.4.2) made it preferable to develop the tool in-house. Regarding the choice between using test case generators (see Section 4.4.1) or writing them manually, test case generators were not considered beneficial for the department. This because the department thought that it would be more cost-effective to write the design specifications as executable test cases directly instead of first writing formal design specifications manually and then generate test cases from them.

Next, the department needed to choose a language to write the test cases in. The major benefits of using standard script languages as for example Visual Basic and JavaScript are that programmers tend to be less error-prone compared to when using system-programming languages (Kaner et al. 2002). Still, the department chose to use C++ as test case language. Firstly, because the developers do not need any training on how to use it since they write the product code in it. Secondly, C++ has the power to handle features that are not included in the actual tool, e.g. it is possible to make direct calls on functions in the product code when the tool has no support for it implemented. Finally, when the test cases are written in the same language as the product code, the developers can take advantage of the programming tools already available (Fewster and Graham 1999), (McGregor 2001).

4.3.2 Test Case Syntax and Output Style

After deciding to develop an in-house Basic Test tool with C++ as test case language, the department chose to use the framework-driven approach when designing the tool, i.e. the tool isolates the component to test from the test cases through wrappers and utility functions (see Section 4.4). Figure 4.2:A shows an example of a test that was implemented in the new Basic Test tool.

In addition, the tool contains several commands for handling component states and for sending and receiving requests. All actions are controlled by the tool and during the execution, the tool compares each sent/received request with its expected result and then logs the result in an XML file. The XML requests are generated from their corresponding DTD's (Data Type Definition).

Figure 4.2:B shows an example of a log output from a test execution. The department chose XML as output format because it is already used as standard data format in

A, Test code example (C++)

```
Tool.startTest("Test1");
  theComponent.startComponent();
  Tool.startTest("Test1:1");
    ToolSender.sendMessage(Request.xml,
                           ExpResult1.xml);
    ToolSender.sendMessage(Request2.xml,
                           ExpResult2.xml);
  Tool.endTest();
  Tool.startTest("Test1:2");
     ToolReceiver.receiveMessage(ExpRes3.xml);
  Tool.endTest();
  theComponent.stopComponent();
Tool.endTest();
```

B, Test result example (XML)

```
<Test name="Test1" status="Failed">
  <Statistics>
       <StartTime>2003:01:01-12.24</>
       <EndTime>2003:01:01-12.25</>
       <NumberExecutedTests>2</>
       <NumberPassedTests>1</>
  </Statistics>
   <Test name="Test1:1" status="Failed">
     <Request1>
        <Result>Ok</Result>
     </Reques1>
     <Request2>
        <Result>Not Ok</Result>
         <ResultFile>ComparisonRes.xml</>
     </Request2>
   </Test>
    <Test name="Test1:2" status="OK">
     <Request1>
        <Result>Ok</Result>
     </Request1>
   </Test>
</Test>
```

Figure 4.2: Basic Test tool - Example

their products and because nowadays, XML is a standard format to which many other tools and parsers easily can be attached. For example, it is easy to develop a GUI that parses the XML data into tree structures of test results (according to the tag structure in the XML). In such a GUI, the developers can monitor their test executions and the managers monitor the test progress.

4.3.3 Adjustments to the Development Process

Just providing a good tool does not ensure successful test automation; a tool only becomes as good as the people using it (Mosley and Posey 2002). Since this tool was to be used with TDD (see Section 4.2.1), the test cases should be written before the code. As earlier mentioned, the department introduced TDD on a component level instead of on a class level, i.e. the developers construct a set of test cases for each component instead of a set of test cases for each class. The test strategy was to capture all inputs and outputs of each component. Thereby, the tests represent the external design of each component. This also led to test cases that could serve as a part of the design documentation, replacing some of the old design (e.g. component specifications). The result of this was a more thorough design (in comparison to plain English, C++ does not leave room for misinterpretations) and that some design time could be saved when being able to remove some of the old design documentation.

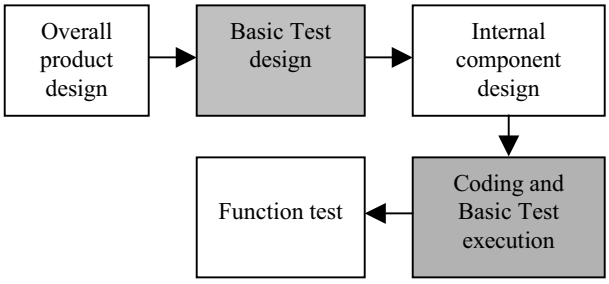


Figure 4.3: Basic Test process at the department

Figure 4.3 displays how the Basic Test concept was incorporated with the process levels at the department. The new activity “Basic Test design” is when the developers construct the components’ test cases and it comprises both implementation and inspection of the test cases.

With a tool and a process for how to use the tool established, there is only one major thing left to put in place: a standard for how to write test cases. Otherwise, the department would end up with spaghetti tests that are hard to understand and maintain leading to that the benefits with test automation would be lost (Fewster and Graham 1999). Test automation is development and the test cases are software programs testing other software programs; therefore, the test cases are subject to the same design and construction rules as the programs to test (Mosley and Posey 2002), (Patton 2001). Thus, the tests should follow ordinary guidelines for structured programming so that they become simple, reusable and easy to understand and maintain. Further, the tests should be independent (Kaner et al. 2002) since this enables the possibility to only execute individual tests when short execution time is crucial. Additionally, when having independent tests, the source of defects can only be within that test, which makes it is easier to locate the source of a defect. In the Basic Test tool, the ’startTest’/’endTest’ constructs (see Figure 4.2) enable such a test independence. Finally, to enable daily regression testing, the tests must be constructed for repeatability, i.e. it must be possible to execute them as regression tests without human interaction (Maximilien and Williams 2003). To be able to track and repair the defects found in regression testing, good naming conventions for the tests are necessary. As can be seen in Figure 4.2, all tests have a nametag which the department used for specifying what feature each test is supposed to verify. Such traceability makes it possible to at all times know the progress of each feature (percent developed/passed).

To ensure that the developers would follow the standard for writing test cases,

someone should inspect all test cases before letting the developers start implementing the product code. Moreover, the inspections must ensure that the tests not just show that the code will work, but also the opposite (Rakitin 2001). To achieve that, adequate test coverage must be obtained, e.g. cover all behavior variants (including error cases and boundary values).

Another challenge the department had to address was to implement the new Basic Test concept on products that already existed since several years and therefore, the components contained a lot of old functionality that did not have such tests. Since developing tests for all old functionality when introducing the tool would be far too costly for a single project to handle, the department chose only to develop tests for new and modified functionality. Henceforth, the strategy was to develop tests for more and more of the old functionality during upcoming projects (i.e. in future releases of the product).

4.3.4 Observations and Lessons Learned

During the implementation and introduction of the new concept, several observations were made and some lessons were learned. This section provides a list of those that were identified by two of the authors of this paper that participated in the target projects and from qualitative interviews with managers and developers at the department after the introduction of the concept.

The department established that:

- Test automation requires high product testability. According to related work, it is the product interfaces that determine the opportunities for test automation (Kaner et al. 2002). The notion of testability is further discussed in Section 4.4.2.

- As also reported in (Mosley and Posey 2002), it is important with a thorough framework design because it makes test case development easier and is robust to future changes.

- Test automation on a component level requires adjustments to the product architecture since component-level test architectures must have a tight integration with their product architectures (also acknowledged in (McGregor 2001)).

- Making people add new tests every time a new defect is found requires continuous remainders and monitoring. Otherwise, developers tend to deliver the bug fixes untested. Furthermore, the authors in (Maximilien and Williams 2003) state that adding new tests for each fault strengthens the test suite.

- Test execution speed is important. Other studies support this experience with the claim that the faster the test execution speed is, the more likely developers will execute the tests themselves (Maximilien and Williams 2003).
- What gets measured gets done (Humphrey 2002), (Rakitin 2001). When the department started the introduction of the new concept for Basic Test, the target projects gave it modest attention. However, when the projects started measuring the progress for number of test cases developed and passed, the usage rates increased.
- Beware of attempts to deviate from the agreed process. When in time-pressure, projects tend to neglect some activities which might give near-time benefits but that might be devastating in the long run (Humphrey 2002). Therefore, such deviations should not be allowed without being agreed on by all involved parties (e.g. the line organization).
- Test-driven development makes people think through the design more instead of rushing to coding directly without knowing what to implement yet.
- When introducing new ways of working, it requires significant efforts to convince people in the organization that the new methods will improve the productivity. If not succeeding to do this, the introduction of the new methods will most likely fail due to resistance among managers and developers to accept the new ways of working (further discussed in Section 4.4.2).
- Test tools easily become the excuse for every problem (Fewster 1993). Reasons for a problem can be in either the test tool, the development environment or in the application under test; still, the Basic Test tool became the scapegoat for most problems since it is in the tool the problems first are discovered no matter where they origin.
- Automated testing requires dedicated resources. As also considered a common fact (Kaner et al. 2002), (Mosley and Posey 2002), test automation cannot be managed as a spare-time activity. In comparison, developing or buying the tool is rather cheap; it is activities such as setting standards for test case writing, teaching users how to write good test cases, and tool maintenance that are costly.
- Benefits from test automation are hard to obtain in the first project release. Other reports support this experience by claiming that upfront costs eliminate most benefits in the first project and benefits from regression testing are usually not realized until the second release (Kaner 1997).

Product version	Expected lead-time difference (development time)
Version X (2003)	-2%
Version X+1 (2003-2004)	-19%
Version X+2 (2004)	-25%

Figure 4.4: Expected lead-time gains with new concept

- Test automation should be introduced in small steps, e.g. as a pilot project to avoid taking unnecessary risks (if something is introduced in the wrong way but only in small scale, the cost of fixing the problem is most likely lower). This advice is also given in the research literature (Fewster and Graham 1999), (Hayes 1995), (Kaner et al. 2002).

- Minimizing maintenance costs is the most difficult challenge in test automation. The test cases must be robust to changes during bug fixes and in new product versions. Since this is hard to achieve, the most common problem in test automation is probably uncontrolled maintenance costs (Kaner et al. 2002).

4.3.5 Expected Lead-Time Gains

This paper does not provide actual results on costs and benefits of the introduced concept. However, preliminary project evaluations indicate significantly decreased fault rates. Further, Figure 4.4 presents the result of a study where all the developers (in a questionnaire) estimated the lead-time difference after they had used the concept in product version X. On average, they estimated that the project lead-time would decrease more and more when using the concept (e.g. 25% in the third project). Also note that the developers did not think that the introduction costs in the first version delayed the project, i.e. they estimated gains already from the beginning.

Another finding (from qualitative interviews with the project managers) was that not only decreased lead-time was the reason for using the concept; they thought that increased progress control (percent test cases executed/completed), and increased delivery precision (due to increased progress control and improved quality assurance) were at least as important.

4.4 Discussion

4.4.1 Related Techniques for Test Automation

The purpose of this section is to relate the techniques used by the department's test automation tool with other techniques used in the software test-automation industry. The presented techniques are described together with some of their pros and cons. However, note that the primary objective of this section is to benchmark the tool, not to perform a technique evaluation.

Capture-replay

The basic concept of capture-replay is that a tool records actions that testers have performed manually, e.g. mouse clicks and other GUI events (that later can be re-executed automatically). Capture-replay tools are simple to use (Beizer 1990), but according to several experiences not a good approach to test automation, since the recorded test cases easily become very hard to maintain (Fewster and Graham 1999), (Kaner 1997), (Kaner et al. 2002), (Mosley and Posey 2002). The main reason is that they are too tightly tied to details of user interfaces and configurations, e.g. one change in the user interface might require re-recording of 100 test scripts (Mosley and Posey 2002).

Script techniques

A powerful approach to test automation is to write some kind of test scripts for the test cases. Script techniques provide a language for creating test cases and an environment for executing them (McGregor 2001). Approaches to scripting varies in several dimensions:

Simple scripts *versus* highly structured scripts: Scripts may vary in complexity from simple linear scripts to keyword driven scripts with conditional/looping functionality and further to real programming languages (Fewster and Graham 1999).

Standard scripts/languages *versus* vendor scripts: Scripting tools either use a standard script programming language (e.g. Visual Basic, C++), or their own proprietary language (vendor script) (Kaner et al. 2002).

Scripts control flow and actions *versus* data-driven testing: This choice is about whether the script or the input data controls the execution. In data-driven testing, an input test data file control the flow and actions (Mosley and Posey 2002).

Standard scripting *versus* framework driven testing: Instead of operating against the product interfaces directly, framework driven testing adds another layer of functionality to the test tool where the idea is to isolate the software from the test scripts.

A framework provides a shared function library that becomes basic commands in the tool's language (Kaner 1997), (Mosley and Posey 2002).

Test Case Generators

Test case generators are the most advanced test automation tools. They exist in several variants: structural generators (generate test cases from the structure of the code); data-flow-generators (use the data-flow between software modules as base for the test case generation); functional generators (generate test cases from formal specifications); and random test data generators (Beizer 1990). Test case generators can generate several test cases fast but are still not always more cost-effective since expected results need to be added manually. Further, the generated test cases/executions of the test cases must also be checked manually to verify that they test the right functionality.

4.4.2 Other Considerations

Before implementing a new concept as the one described in this paper, both the organization and its products must fulfill some prerequisites; otherwise, the risk for failure is substantial. This section describes a few prerequisites in relation to the situation at the department.

Maturity of the Organization

First, the development process that the developers follow needs to be mature enough. If the process is poor, test automation will not help (Fewster and Graham 1999), (Kaner et al. 2002). Preferably, the test automation effort should be easy to adapt to current practices; if the change is too great, the risk for resistance among the developers increases (Johnston 1993). Furthermore, if the developers do not want to work as directed, they will not (Humphrey 2002). At the department, the developers were aware of that neglecting Basic Test results in increased verification lead-time. Therefore, they were open to improvements in Basic Test. Second, the managers must be committed to the new methods because it is they that have to grant the upfront costs with introducing test automation (that most likely will impact short-term budgets and deadlines negatively (Berwick 1993)). Test tool costs is just the tip of the iceberg (Fewster and Graham 1999), (Kaner et al. 2002).

Maturity of the Products (Testability)

A product with high testability provides interfaces that are easy to develop test cases for, robust to changes, and whose data is easy to represent in test cases together with

expected outputs that the test execution tool automatically can verify against the received outputs. The more testable the software is, the less effort developers and testers need to locate the defects (McGregor 2001). For example, the Basic Test tool was significantly cheaper to build when the products had a common communication interface to simulate.

4.5 Conclusions

The Ericsson department introduced a new test automation tool incorporated with an alternate approach to Test-Driven Development (TDD), i.e. TDD on a component level where the interfaces comprise socket connections exchanging XML data instead of classes and methods. With such an approach to test-driven development, robust and uniform component interfaces make test automation easier.

The main characteristic of the tool is that it uses C++, the same standard programming language as the developers write the product code in, because the developers are already familiar with it, it is more powerful than a script language, and the developers can take advantage of programming tools already available (e.g. compilers and debuggers).

Regarding process support, the software development department introduced the concept TDD to support the tool mostly because:

- When writing the test cases before the code, testing really happens.
- The test cases drive the design and make it more straightforward, i.e. with an explicit scope and no gold plating.
- TDD moves fault detection earlier in the development process when the faults are cheaper to find.

There are several aspects to consider when implementing test automation and TDD. Section 4.3.4 lists the most important ones (in form of observations and lessons learned).

The developers that have used the concept have estimated that the project lead-time will decrease more and more for each new project version that uses it (see Section 4.3.5). Further, preliminary project evaluations indicate significantly decreased fault rates from the introduction of the new concept.

Chapter 5

Case Study Results from Implementing Early Fault Detection

Lars-Ola Damm and Lars Lundberg

Abstract

For many software development organizations it is of crucial importance to reduce development costs while still maintaining high product quality. Since testing commonly constitutes a significant part of the development time, one way to increase efficiency is to find more faults earlier when they are cheaper to pinpoint and remove. This paper presents empirical results from introducing a concept for earlier fault detection. That is, an alternative approach to test-driven development which was applied on a component level instead of on a class/method level. The selected method for evaluating the result of introducing the concept was based on an existing method for fault-based process assessment and was proven practically useful for evaluating fault reducing improvements. The evaluation was made on two industrial projects and on different features within a project that only implemented the concept partly. The result evaluation demonstrated significant improvements, e.g. every invested hour gave 4.8 and 7.3 hours profit respectively in the two studied project.

5.1 Introduction

Avoidable rework is commonly a large part of a development project, i.e. 20-80 percent depending on the maturity of the organization (Shull et al. 2002). Having many faults left to correct late in a project leads to higher verification costs. That is, several studies demonstrate that faults are cheaper to find and remove in earlier stages of projects (Boehm 1983), (Shull et al. 2002). Further, fewer faults leads to improved delivery precision since software processes become more reliable when most defects are removed in earlier phases (Tanaka et al. 1995).

This paper presents the result of the implementation of a concept for achieving earlier fault detection in two commercial product development projects at a software development department at Ericsson AB. In order to achieve this, the department wanted to develop a framework for automated component testing. Chapter 4 proposed such a framework based on component-level test automation where the test cases are written before the code. That is, an alternative approach to Test-Driven Development (TDD) (Beck 2003). The purpose of this paper is to provide results from implementing the proposed concept in two commercial projects. Thereby, the primary research question is as follows:

What are the costs and benefits of introducing a framework for component-level test automation and Test-Driven Development (TDD) in an industrial setting?

When implementing such changes in an industrial setting, it is not as easy to evaluate the result as for controlled research experiments. That is, in commercial projects, it is likely that several other variables than those that were part of the planned change are affected, e.g. new project members and increased organizational maturity. Further, many SPI frameworks such as CMM and ISO 9000 advocate implementing many improvements at the same time (Conradi and Fuggetta 2002). Therefore, it is for example hard to relate cost and lead-time changes to certain improvement actions.

However, since the primary output of testing is found faults, a possible approach to such a result evaluation is to look at differences in fault distributions. However, in order for such an approach to be useful, the following two criteria should be met:

1. The approach should be able to quantify the obtained benefits of the improvement; just classifying faults into categories is not enough.

2. Since several differences between projects might affect the fault distributions, the approach should be able to relate differences in fault distributions to a certain change.

Classification of faults from their causes, e.g. root cause analysis is a very common approach to fault classification (Leszak et al. 2000). However, neither root cause analysis, nor other fault classification approaches such as ODC triggers (Chillarege et al. 1992) are applicable in this context since they are not feasible for cost benefit analysis. Further, there exist techniques such as 'Six Sigma' (Biehl 2004) that evaluate quality improvements by measuring differences in fault distributions in relation to product size. However, these cannot relate differences in fault distributions to a certain change.

Another possibility is as described in Chapter 3 to measure the amount of faults that should have been found in an earlier phase, i.e. 'faults-slip-through'. Although this method was originally intended for process assessment, it could also be used for quantifying the benefits of an improvement that should result in finding more faults earlier. That is, the method also measures the number of hours of avoidable fault costs in the assessed projects by multiplying the number of faults-slip-through with average fault costs in different test phases. Since the method also can compare fault costs in relation to phases, it should also be possible to determine if the differences in fault costs matches the phases where an improvement should have had an effect. Thus, the method satisfies the criteria for being able to evaluate the primary research question. Consequently, the second research question is:

How can costs of faults-slip-through be used for quantifying the benefits of software test process improvement?

This paper is organized as follows. After an overview of related work in Section 5.2, the proposed method is described in Section 5.3. Then the practical applicability of the method is demonstrated when evaluating the primary research question in Section 5.4. After that, a discussion regarding the value, validity and applicability of the results is provided in Section 5.5. Finally, Section 5.6 concludes the work

5.2 Related Work

Before describing the method and case study results, this section maps the implemented concept and the selected result evaluation method to related research work.

5.2.1 Early Fault Detection and Test Driven Development

Unit testing and inspections are common techniques for detecting fault early. Several widely used techniques for testing units based on their internal structure exist, e.g. class testing, random testing and partition testing (Beizer 1990). Some unit test techniques implement assertions directly in the product code. Test-Driven Development (TDD) is

one technique that has such an approach (Beck 2003). Commonly, tools such as JUnit or CPPUnit are used together with TDD (Beck 2003). The main difference between TDD and a typical test process is that in TDD, the developers write the tests before the code. A result of this is that the test cases drive the design of the product since it is the test cases that decide what is required of each unit (Beck 2003). Therefore, TDD is not really a test technique (Beck 2003), (Cockburn 2002); it should preferably be considered a design technique. In short, a developer that uses traditional TDD works in the following way (Beck 2003):

1. Write the test case
2. Execute the test case and verify that it fails as expected
3. Implement code that makes the test case pass
4. Refactor the code if necessary

Within the area of component testing, several techniques are suggested and implemented (Gao et al. 2003). Although, component testing more or less could be managed as ordinary black box testing, some of the more advanced techniques attempt to incorporate the tests in the components more like how unit testing is performed, e.g. using concepts such as XML based component testing (Bundell et al. 2000) or built-in tests (Edwards 2001).

Regarding the combination of automated component testing and TDD, little experience exists. TDD has been successfully used in several cases within agile development methods such as eXtreme Programming (XP) (Beck 2003), (Rasmusson 2004). However, the applicability of TDD has so far not been fully demonstrated outside of that community. Teiniker et al. (2003) suggest a framework for component testing and TDD (Teiniker 2003). However, as for the case in our own previous publication described in Chapter 4, no practical results regarding the applicability of the concept exist. Further, as opposed to our solution, the framework described by Teiniker et al. (2003) focus on model driven development and is intended for COTS (Commercial Of The Shelf) component development.

5.2.2 Evaluation of Fault-Based Software Process Improvement

Within the area of Software Process Improvement, several techniques for cost and benefit estimations have been developed (Boehm 1983), (Paulish and Carleton 1994), (Van Solingen 2004). Commonly, these techniques look at several aspects such as costs, benefits, and quality aspects. Further, some techniques evaluate quality improvements by

measuring differences in fault distributions in relation to product size, e.g. 'Six Sigma' (Biehl 2004). Other approaches for differences in fault distributions in relation to fault costs also exist (Hevner 1997). A similar approach is also taken in the method that provides the basis of the method that is proposed in this paper, i.e. the method described in Chapter 3. However, these approaches focus on cost assessment and fault distributions; they do not include benefit estimations. Further, the approach to fault classification used in (Hevner 1997) is as described in Chapter 3 at least not applicable in organizations such as the one presented in this report since it focuses on fault introduction phase instead of fault belonging phase.

5.3 Method

The method part of this paper starts with a background on the organization and case study. Then, Section 5.3.2 presents the result evaluation method, i.e. how to determine the costs and benefits of the concept.

5.3.1 Background

This section includes an overview of the studied organization and a background on the motivation for and layout of the conducted case study.

Organizational Background

The results of this paper are obtained through a case study at a software development department at Ericsson AB. The department runs several projects in parallel and consecutively. The projects develop new functionality to be included in new releases of existing products that are in full operation as parts of operators' mobile networks. Each project lasts about 1-1.5 year and has on average about 50 participants. Further, the products are built on a shared platform, i.e. as a product-line architecture where the platform provides a component-based architecture and a number of platform components. Thus, the products build components on the same component architecture and reuse platform components when needed. The components are built in C++ except for a smaller Java-based graphical user interface. Each component contains about 5-30 classes and the components communicate mainly through a common socket connection interface and the data sent between the components is in XML format. Further, the components are verified in three steps: Basic Test, Function Test, and System Test. Since the products constitute smaller part of large mobile networks, they are rather difficult to verify. Therefore, verification against other product nodes in the mobile

networks is a large part of System Test. Before releasing the product globally, initial field tests are performed together with a trial customer. Finally, only faults found as of Function Test are stored in a fault reporting system, i.e. faults found earlier are not reported in a written form that can be post-analyzed. Further, some of the reported faults are not real product faults because they were false positives or they did not affect the operability of the products, e.g. opinion about function, not reproducible faults, and documentation faults. Therefore, these faults were excluded in the case study presented in this paper.

Case study background

As mentioned in the introduction, the implemented solution was the result of a previously identified opportunity for improved test efficiency by finding more defects earlier, i.e. as described in Chapter 3. Additionally, the study determined how much more expensive faults are to find in later phases, which made it possible to quantify the improvement potential of finding more faults earlier, e.g. in the two studied projects, the fault cost during testing could as described in Chapter 3 be decreased by up to 85 and 86 percent by finding those faults before delivery to the test department instead.

Besides identifying in what phase improvements should be directed, the previous study also determined what activities that needed to be addressed, e.g. better tool and process support. In particular, it was concluded that a large reason for insufficient component testing before delivery to integration tests was as described in Chapter 4 due to the deadline pressure that commonly occur shortly before the deliveries. During such time pressure, people tend to deliver the code with lesser quality assurance hoping that they have not made any mistakes. To address this problem, a central part of the process change was to introduce component-level Test-Driven Development (TDD). The reason why TDD could make developers test more is that when writing the test cases before the code it is more likely that the written tests are executed before delivery (Cockburn 2002). From this analysis, Chapter 4 proposed a concept consisting of component-level test automation and TDD.

The concept was as described in Chapter 4 based on an in-house implemented tool that could send requests on a component's interfaces, i.e. simulating the surrounding components. Commercial tools for TDD such as CppUnit were not applicable since they operate on a class level. The implemented tool provided library routines to use when writing the tests. Then it could control and monitor the execution of the implemented test cases, and finally analyze if the tests passed and log the result in an XML file.

Applying TDD on a component level meant that some differences in comparison to traditional TDD were required. The test cases were not developed for each

class/method but instead for each component interface. Additionally, since the department wanted to use TDD as a part of component design, the test cases were as further described in Chapter 4 treated as an executable specification that replaced some of the old design documentation.

The concept was implemented in two new product releases that were developed in parallel and this paper evaluates the impact the concept had on the two projects. In practice, this means that the case study focuses on comparing the two projects that implemented the suggested solution with their predecessors. However, in one of the projects, the new solution was not implemented fully. That is, only some of the features implemented in that project used the new concept. Thereby, the evaluation also quantified differences between these features in comparison to the features that did not implement the new concept.

5.3.2 Result Evaluation Method

This section describes the selected method for evaluating a software process improvement implementation in an industrial setting. The essence of the method is to measure benefits in form of decreased fault costs and then measure the Return Of Investment (ROI) by comparing the benefits with the investment costs, both as total ROI and ROI in relation to a certain phase where an improvement has been applied.

Investment Cost Estimation

Implementing new processes and tools in an organization inevitably involves up-front costs such as tool acquisition and training. Such costs are often a one-time investment and do not necessarily pay off directly in the first project. More importantly, changed ways of working might affect project costs also in the long run. For example, in the case study in this paper, additional work with test-case writing could be expected from introducing TDD. However, at the studied department, it was hard to measure such a cost because the department did not just add TDD as a new activity but rather as a replacement to other activities. That is, using TDD made test case writing a design activity, which then replaced other design documentation. Further, as described in Chapter 4, making a more thorough design made the coding phase faster. Based on estimations by developers using the TDD-based new concept, it could instead be assumed that the additional test design cost would not result in longer total development time due to the benefits obtained already during coding. In fact, the only costs that really affected the projects were training costs and initial tool usage problems. Therefore, the cost evaluation in this paper just measured these costs and then compared them with the obtained benefits. Further, tool development costs were not included in the cost calculations

since they did not affect the projects. That is, the tool was developed outside of the project budgets and was considered as a line investment that should pay of in the long run. Nevertheless, the tool development cost was about six man-months.

In the case study evaluation in this paper, the measured investment costs (training and initial tool usage problems) were obtained through a questionnaire where each developer estimated how much of the reported time they put on different activities (where training and initial tool usage problems where two of those).

Faults-Slip-Through and Fault Cost Assessment

As suggested in the introduction, the benefit evaluation of the introduced concept should be based on measures obtained when using an existing method described in Chapter 3, i.e. the measures faults-slip-through and Avoidable Fault Cost (AFC). The method consists of the following three steps:

1. Determine which faults that could have been avoided or at least found earlier
2. Determine the average cost of finding faults in different phases.
3. Determine the AFC from the results in (1) and (2).

As previously mentioned, the introduced measure for determining step one is called 'faults-slip-through', i.e. whether a fault slipped through the phase where it should have been found. When applying the faults-slip-through measure, the organization must therefore define which faults should be found in each phase, e.g. based on what should be tested in each phase. For example, performance and load faults may belong to the System Test phase and are thereby not slip troughs when found there. Although Faults-Slip-Through (FST) in this method primarily was intended as input to step three (as described further below), the measure may also be used as a stand-alone process measure. The studied organization does this by measuring total percent faults-slip-through to Function Test (FT) and System Test (ST) respectively. In the result presentation of this case study as described in Section 4, these measures are included in the project comparison, presented as 'FST to FT' and 'FST to ST'.

Regarding step two, the cost-increase varies significantly depending on the maturity of the development process and the product environment, e.g. desktop software versus embedded systems (Veenendaal 2002). Therefore, the average fault cost in different phases needs to be determined explicitly in the environment where the improvement potential is to be determined. This measure could either be obtained through the time reporting system or from expert judgments, e.g. a questionnaire where the developers/testers that were involved in the bug-fix give an estimate of the average cost. The

estimates used in this case study were obtained through the expert judgments obtained in the previous study as described in Chapter 3 (see also Figure B.1 in Appendix B).

The third step, i.e. the AFC measure (named improvement potential in Chapter 3) is determined by calculating the difference between the cost of faults in relation to what the fault cost would have been if none of them would have had slipped through the phase where they were supposed to be found. Equation 5.1 describes the formula to use for calculating the AFC. Further, Appendix A provides an example calculation (with fictitious values) of how to apply the formula. The next section describes how such AFC calculations can be used as input for benefit estimation of an improvement, i.e. in form of decreased AFC between projects.

$$AFC = (Nr\_faults\_belonging * Average\_cost\_of\_faults_{phase\_found}) - (Nr\_faults\_belonging * Average\_cost\_of\_faults_{phase\_belonging}) \quad (5.1)$$

Return On Investment (ROI)

From the investment costs and the benefits in form of decreased fault costs (as described in the previous two sections), it is possible to estimate the ROI of an implemented change. For the benefit part of ROI, it would be natural just to apply the AFC calculation above to compare the projects. However, since the amount of faults commonly increases with project size, this variable should also be included in the AFC calculation. This could simply be achieved by calculating the AFC as a cost ratio in comparison to the project size. Project size can be measured in several ways, e.g. as product size (LoC, function point analysis) or as cost of development. When relating the AFC values and project sizes before and after an implemented improvement, the AFC improvement can be calculated as a relative ratio. Equation 5.2 provides the formula for such a calculation.

$$Improvement\_ratio = \frac{AFC_{WithoutChange}/Projectsize_{WithoutChange}}{AFC_{WithChange}/Projectsize_{WithChange}} \quad (5.2)$$

From the obtained AFC improvement ratio, it is then also possible to calculate how much lesser the AFC was in project 2 in comparison to if the improvement had not been implemented. In fact, such a calculation estimates the actual benefit from implementing a change. As illustrated in Equation 5.3, the benefit can be estimated by calculating what the AFC would have been without the improvement (improvement*$AFC_{WithChange}$) and then compare this cost with the actual AFC.

$$Benefit = (Improvement\_ratio * AFC_{WithChange}) - AFC_{WithChange} \quad (5.3)$$

Finally, as can be seen in Equation 5.4, ROI is calculated as a ratio of estimated benefit with investment costs of an improvement. Thus, the ROI is presented as a value between 0-X (Van Solingen 2004). That is, when interpreting a ROI value, every invested hour gave X hours profit, i.e. all values above zero means a positive return. Note that when aiming at exact ROI measurements other factors such as time (e.g. Net Present Value) and risk discount should also be included (Erdogmus et al. 2004). However, these factors were discarded because they are hard to account for and the method suggested in this paper was intended to be pragmatic. That is, since ROI calculations for software process improvement usually do not need to be exact; pragmatic estimations that are easy to obtain but still are good enough for decision making are preferred (Van Solingen 2004).

$$ROI = \frac{Benefit - Investment cost}{Investment cost} \tag{5.4}$$

Since the underlying method for benefit estimation is based on fault costs in different phases (see Section 5.3.2 and Appendix A), it is also possible to compare obtained improvements for certain phases. For example, in this case study, the AFC ratio for faults slipping through from implementation should decrease when implementing an improvement in that phase. The benefit estimation for the implementation phase is made with the same formulas as when calculating the total difference, i.e. by replacing the total AFC with AFC of the implementation phase in equations 2-4 as describe above. The AFC change related to the implementation phase is calculated through this method and included in the result calculations in Section 5.4 and Appendix B, presented as Improvement ratio (Impl.) and ROI (Impl.).

5.4 Result

This section evaluates the effect of introducing the previously described framework for component-level test automation and TDD in an industrial setting. This is achieved by applying the method described in Section 5.3. The evaluation was performed in two dimensions. First, two projects were compared against their respective predecessors. Then, features within one of these projects were compared divided after whether the new concept was implemented for that feature or not.

5.4.1 Comparison against baseline projects

When comparing projects against their predecessors, the benefits are first presented as differences in faults-slip-through distributions and then the ROI is presented both

as a total and in relation to the implementation phase where the improvement was implemented.

Faults-Slip-Through

Table 5.1: Faults-slip-through - Comparison against baseline projects

	Release X	**Release X+1**	**% Improvement**
Product A FST to FT	69%	65%	5.4%
Product A FST to ST	77%	55%	29%
Product B FST to FT	80%	71%	11%
Product B FST to ST	77%	54%	31%

Since two projects implemented the new concept, the obtained measures are presented for those two projects and their respective predecessors. Table 5.1 presents the faults-slip-through measures calculated as described in Section 5.3.2 That is, the percent Faults-Slip-Through (FST) to Function Test (FT) and System Test (ST) for each project is displayed in the first two columns. For example, in release X of product A, 69 percent of the faults reported during FT should have been found earlier. Further, as can be seen in Table 5.1, the FST decreased in both projects where the new concept was introduced, e.g. in Product A, the FST to ST was improved by 29 percent.

Return Of Investment (ROI)

Figure 5.1 presents the improvement ratio and ROI as calculated in Appendix B (by applying the formulas described in Section 5.3.2). In the figure, the improvement ratio was high in both projects, i.e. in relation to project size, the avoidable fault cost decreased about 2-3 times after the improvement. Further, the improvement ratio was at least as high when just including the implementation phase. This means that the improvement occurred due to the changes in that phase. In fact, since the improvement was even higher for the implementation phase than the total value, the improvement ratio would have been negative without the improvement in that phase, i.e. because the number of faults in other phases actually increased.

Further, the ROI was significant in both comparisons, i.e. every invested hour gave about 4-5 hours profit in Product A and about 6-7 hours profit in Product B. However, note that as described in Section 5.3.2, only investment costs that directly affected the projects were included in the ROI calculation. Otherwise, the ROI would not have been as high directly in the first projects.

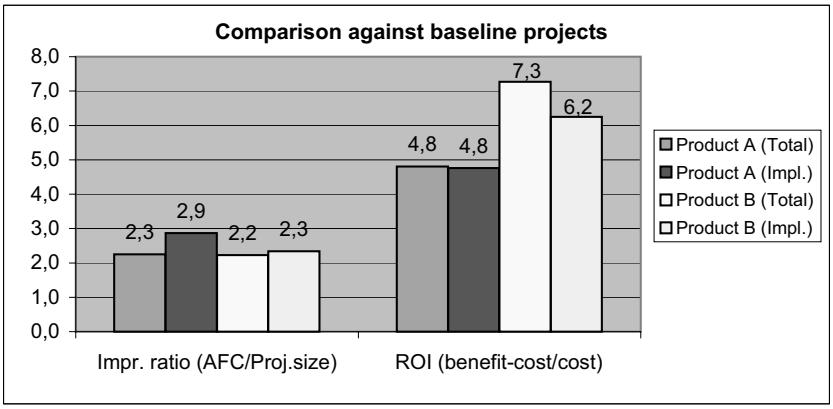


Figure 5.1: Improvement ratio and ROI when comparing against baseline projects

5.4.2 Comparison between features within a project

By applying the same method as in Section 5.4.1, the comparison between features was made for the second release of Product B, i.e. the project were the new concept was partly introduced.

Faults-Slip-Through

As can be seen in Table 5.2, the faults-slip-through distributions were similar when comparing features within a project as when comparing between projects in Section 5.4.1, i.e., the measures indicate similar improvement rates from introducing the new concept. The only notable difference was that the faults-slip-through rates both without and with the improvement were slightly lower than when comparing between project releases. The probable reason for this was that the increased attention to find more faults earlier to some extent also affected the whole project, not only the features that used the new concept.

Table 5.2: Fault statistics – Comparison between features within a project

	Without	**With**	**% Improvement**
Product B FST to FT	73%	67%	8.5%
Product B FST to ST	57%	46%	19%

Return Of Investment (ROI)

Using the same formulas as when comparing against baseline projects, Figure 5.2 presents the resulting improvement ratios and ROI values, obtained from the calculations in Appendix B (Figure B.2). Although Fig. 5.2 shows that the improvement ratios and ROI also were positive in this comparison, the improvement rates were lower, especially regarding ROI. The reason for lower improvement ratios was most likely the same as for the decreased faults-slip-through, i.e. increased attention to early fault detection affected all features although only some of them explicitly implemented the new concept. Further, the improvement ratio for the implementation phase was in this comparison unexpectedly low in comparison to the total ratio, i.e. 1.9 versus 2.3. This indicates that not only changes in the implementation phase affected the total improvement. The major identified reason for this was that one of the features that did not use the new concept, i.e. a new installation system, was tested separately in the phase Basic Node Test (BNT) and generated a significant faults-slip-through from that phase (see the BNT columns in the right section of Figure B.2 in Appendix B). This demonstrates the need for a measure that not only evaluates the total improvement; only a part of it (1.9) can be contributed to the implemented concept.

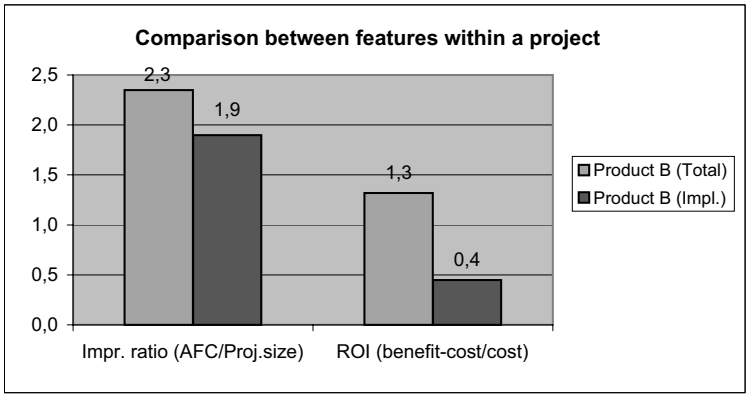


Figure 5.2: Improvement ratio and ROI – Comparison between features within a project

Further, the reason for the large ROI difference (i.e. 0.4 and 1.3 versus 4-7 when comparing projects) was that the number of hours that the AFC decreased was significantly less when comparing features within the project in relation to when comparing with the previous project. This effect was expected because although the investment costs were the same, comparing whole projects involves higher AFC values than when

making a comparison within a project (see Appendix B). However, note that a ROI as low as 0.4 is still a profit.

5.5 Discussion

This purpose of this section is to discuss the major findings in this case study and the method used for obtaining them. First, the value and validity of the presented results are discussed, i.e. what do they mean in practice and are the trustworthy? Next, the usefulness of the defined and applied method is discussed; that is, does it really work in practice?

5.5.1 Value and Validity of the Results

Result Value

The case study results in Section 5.4 show that the introduced concept had a significant ROI (from decreased fault costs). However, ROI is only one of the key factors for evaluating the results of process improvement initiatives. In order to stay competitive, companies must deliver high quality products on time and within budget. Although the development cost was very important, quality, lead-time, and delivery precision were considered as the most important factors. Therefore, an evaluation of what effect the implemented concept had on these factors was of interest.

Since it in the conducted case study was hard to get any quantitative data for the other factors, only a qualitative evaluation of the effect of the fault cost decrease on the other factors was made. That is, the information was retrieved through discussions with managers and test coordinators. Still, as can be seen in the following list, the analysis could at least provide some indications of the total result.

Quality. Since the decreased fault cost occurred due to decreased fault-slip-through rates, it is likely that the amount faults in the delivered product decreased and thereby the quality increased. One can at least be quite certain that the quality did not decrease.

Lead-time. Since a single severe fault might affect the lead-time significantly, it is hard to make a direct relation between decreased fault costs and required lead-time. However, in the case study, a fact is that if the amount of faults that slip through to later phases decrease significantly, the amount of required bug-fix deliveries to those test phases decreases. Since test leaders in the studied projects stated that the amount of bug-fix deliveries is the most important factor for required test lead-time, the new concept most likely decreased the lead-time.

Delivery precision. As for lead-time, single severe faults might affect delivery precision significantly. However, if few faults remain to be found in later test phases, the amount of remaining test time is easier to estimate and the project is more likely to be able to meet the set schedules. Managers actually stated that the motivation for decreasing faults-slip-through rates was foremost not about decreasing costs but instead about to decrease the risk to have to ship the product late. Although the new concept most likely increased the delivery precision, it was not possible to quantify since there are so many other factors that might affect the delivery precision.

To summarize the discussion in this section, the concept most likely improved the other factors, at least it did not become worse. Finally, there might also be other positive effects of the concept that are hard to quantify, e.g. increased organization maturity, and increased employee motivation (Van Solingen 2004). These effects make a company more competitive and employees are more likely to stay.

Validity Threats to the Results

When conducting an empirical industry study, the environment cannot be controlled to the same extent as in isolated research experiments. In order to be able to make a correct interpretation of the results presented in Section 5.4, one should be aware of threats to the validity of them. As presented below, the main validity threats to this case study concern conclusion, internal, and external validity (Wohlin et al. 2003). Construct validity is not relevant in this context since the case study was conducted in an industrial setting.

Conclusion validity. Since the results were calculated from the faults-slip-through measure and the average fault cost measure, the accuracy of the results are only dependent on the accuracy of these underlying measures. Regarding these measures, the threats to conclusion validity are as follows.

First, when determining which phase each fault belonged to, the department must have a common view on which phase each fault should be found in. That is, managers and developers should together agree on which fault types that should be considered as faults-slip-through and not. At the studied department, this was managed by having workshops where checklists for how to estimate fault-slip-through were developed. However, continuous monitoring and training are required in the first projects in order to ensure that everyone have the same view on how to make the estimations.

Further, regarding the average fault cost in different phases, the result was obtained through expert judgments and therefore, the estimations might not exactly reflect the reality. However, this was minimized by asking as many 'experts' as possible, i.e. although there might be deviations, the results were good enough to measure the improvement potential from and hence use as basis for decisions. Nevertheless, in the

future, direct fault cost measures should be included in the fault reports so that this uncertainty is removed.

Internal validity. In the case study presented in this paper, all faults were post-classified by one researcher, which thereby minimized the risk for biased, or inconsistent classifications. Additionally, to ensure that the researcher would not also be biased, a test manager and a test leader afterwards analyzed the classified faults and validated that the classification was correct.

Another threat to internal validity is whether certain events that occurred during the studied projects affected the fault distribution, i.e. events that the researchers were not aware of. This was managed through workshops with project participants where possible threats to the validity of the results were put forward. Further, since two projects were measured and features within one of the projects also were internally compared, the likelihood of special events that affected the results without being noticed decreased. Additionally, usage of phase differentiation also increased the validity, i.e. the evaluation also isolated the effect on the implementation phase where the new concept was implemented. Thereby, the only remaining threat is other changes within the implementation phase. However, no other improvements were implemented in this phase and since comparisons were made both between and within projects, the risk for unnoticed changes became very small.

External validity. In this case study, the results are in overall not fully generalizable since they dependent on the studied department having certain products, processes, and tools. Nevertheless, the results are generalizable within the context of the department and other similar contexts.

5.5.2 Applicability of Method

As indicated in the introduction, the results of process and tool improvements are not very easy to quantify in commercial projects since many planned and unplanned changes occur between and within projects. Still, it is in such real environments new processes and techniques must be evaluated before practitioners are convinced that they will work. The method described in Section 5.3 can quantify the effect of fault-related software process improvement efforts in an industrial setting. However, from the experiences from using the method, there might be discovered implications or limitations to consider. Since the method relies on two basic measures, i.e. faults-slip-through and average fault cost, these measures are discussed in this section.

Faults-slip-through

Experiences from using faults-slip-through indicate that it is a powerful measure in terms of the usefulness of the results it provides. Additionally, there seem to be other positive empirical research results from using the same measure (see Section 5.3.2). Nevertheless, it does still not seem to be used widely in practice. The reason for this is most likely the same as when introducing many other new techniques, i.e. it is not a bulletproof recipe; it must be applied in the right way in order to be successful (Mosley and Posey 2002). First of all, the faults-slip-through definition must be tailored for the specific organization in order to provide meaningful results. That is, if the measure only relies on subjective judgments, no one will trust it. Further, the measure must provide comparable results that for example can be used in process improvement programs. That is, if the measure for example is applied as fault introduction phase, the results will probably always be at least 95 percent faults-slip-through since almost all faults technically can be found earlier. Such an interpretation makes the measure more or less useless for quantitative evaluations.

Further, regarding the definition of the measure, it is important that the measure is project independent and instead defined after organizational goals. That is, if the definition of the measure differs between each project, the projects cannot be adequately compared. Further, for increased and commitment and correctness, the definition should be developed together with the people that are developing the product, e.g. through workshops.

Finally, organizations implementing the measure should be aware of that its usefulness is not only about the quantitative data it can provide to managers. It also becomes a facilitator for discussing and agreeing on what to test when, e.g. improvement of test strategies. This implies that the measure could serve as a key driver for test process improvement.

Average fault cost

Although, research has provided several results on how much more faults cost in later phases, the measure should be determined specifically in the organization where to be applied since the average fault costs vary significantly depending on the maturity of the organization and the types of systems the organization develops (Shull et al. 2002), (Veenendaal 2002). The measure could either be obtained through the time reporting system or from expert judgments. Using expert judgments is a fast and easy way to obtain the measures; however, in the long-term, fully accurate measures can only be obtained by having the cost of every fault stored with the fault report when repairing it. That is, when the actual cost is stored with each fault report, the average cost can

be measured instead of just being subjectively estimated. Further, faults might result in more than just the actual cost of finding and fixing them, e.g. additional bug-fix deliveries or regression tests. Therefore, these costs should also be accounted for when possible. Finally, an advantage with the possibility to measure the fault cost in different ways is that organizations can choose between the 'quick and dirty' solution (i.e. expert judgments) and more exact measurement methods cost more.

5.6 Conclusions and Further Work

This paper presented results of industry experience from implementing a concept for early fault detection, i.e. automated component testing and TDD. The case study comprised four projects, i.e. the concept was implemented in two projects, which were compared against their respective predecessors. Additionally, since one of the projects only implemented the concept partly, features within that project were also compared against each other. From this research setting, the paper set out the following two main research questions to address.

One research question addressed how costs of faults-slip-through can quantify cost and benefits of software test process improvement. The paper proposed and demonstrated the applicability of such a method in an industrial setting and it also demonstrated the usefulness of the results that were provided. That is, the method could for example determine to what extent the obtained results occurred due to the implemented concept, i.e. by excluding possible effects of other changes in other phases.

The purpose of the other research question was to apply the selected result evaluation method on the case study described in this paper. The result was that the implementation of the new concept had significant improvement rates in form of decreased faults-slip-through rates (5-30 percent improvement) and decreased fault costs, i.e. the Avoidable Fault Cost (AFC) was on average about 2-3 times less than before. Further, the phase where the improvement was directed constituted a significant proportion of the decreased AFC. Additionally, in our way of measuring costs and benefits, the Return On Investment (ROI) was positive in the projects were the concept was implemented, e.g. every invested hour gave 4.8 and 7.3 hours profit in the two studied project. This shows that the investment was really beneficial.

Further work includes replicated studies of other ongoing projects that currently use the concept. Further, possible quantitative methods for evaluating effects on lead-time and delivery precision would be of interest to investigate. However, due to the complex nature of commercial projects, such methods can only be based on estimations. That is, to measure the effect a certain process change had on these factors is more or less impossible in an industrial setting.

Chapter 6

Activity-Oriented Process Assessment using Fault Analysis

Lars-Ola Damm and Lars Lundberg

Abstract

Successful software process improvement depends on the ability to analyze past projects and determine which parts of the process that could become more efficient. One source of such analysis is the faults that are reported during development. This paper proposes how a combination of two existing techniques for fault analysis can be used to identify where in the test process improvements are needed. This was achieved by classifying faults after which test activities that triggered them and which phase the faults should have been found in. The practical applicability of the proposed method was demonstrated by applying it on an industrial software development project at Ericsson AB. The obtained measures resulted in a set of quantified and prioritized improvement areas to address in consecutive projects.

6.1 Introduction

Avoidable rework is commonly a large part of a development project, i.e. 20-80 percent depending on the maturity of the organization and the type of development projects (Shull et al. 2002), (Veenendaal 2002). Several studies demonstrate that faults are cheaper to find and remove in earlier stages of projects (Boehm 1983), (Shull et al. 2002). Further, fewer faults leads to improved delivery precision since software processes become more reliable when most defects are removed in earlier phases (Tanaka et al. 1995). Therefore, there is a need for identifying process improvements that can decrease the fault latency.

A software development department at Ericsson AB develops software services for the mobile network. In order to stay competitive, they run a continuous process improvement program where they regularly need to decide where to focus the current improvement efforts. However, as considered a common reality in industry, the department is already aware of potential improvement areas; the challenge is to prioritize the areas to know where to focus the improvement work (Wohlwend and Rosenbaum 1993). Further, if a suggested improvement can be supported with quantitative data, it becomes easier to convince people to make changes more quickly (Grady 1992).

A previously suggested way to guide managers in improvement prioritizations is to measure the amount of faults that should have been found in an earlier phase, i.e. 'Faults-Slip-Through' (FST). That is, as described in Chapter 3, the faults are classified according to whether they slipped through the phase where they should have been found. This method is good for identifying phases that should be improved but a need for an extension was identified when using it in practice. That is, a test manager at the Ericsson department stated: "I would not only like to know which phases that need improvements; I also want to know which activities in each phase that should be improved".

A commonly used approach for identifying activities that need improvements is classification of faults from their causes, e.g. root cause analysis (Leszak et al. 2000). Although root cause analysis can provide valuable information about what types of faults the process is not good at preventing/removing, the technique is cost intensive and therefore not easy to apply on larger populations of faults. Further, root cause analysis primarily identifies improvements during design and coding. A possible fault classification approach that has been suggested to make fault analysis easier is Orthogonal Defect Classification (ODC) (Chillarege et al. 1992). ODC focuses on two types of classifications for obtaining process feedback, i.e. fault type and fault trigger classification. Fault type classification can provide feedback on the development process whereas fault triggers can provide feedback on the test process (Chillarege et al. 1992). ODC fault trigger classification appeared to be especially promising to apply at

the Ericsson department because they can identify which test activities that need improvements. The purpose of this study is to determine how ODC fault triggers can be used to improve the FST concept, i.e. locate activities that cause FST. Consequently, the research question of this paper is:

How can ODC fault triggers be used to improve the FST concept?

The outline of this paper is as follows. Section 6.2 provides an overview of related work and then Section 6.3 describes the suggested method for answering the specified research question. As described in Section 6.4, the applicability of the method suggested in this paper was demonstrated through a case study in an industrial project. The case study results are in Section 6.5 followed by a discussion on practical experiences and possible validity threats. Finally, Section 6.6 concludes the work.

6.2 Related Work

Within software process improvement, one of the most widely used models is the Capability Maturity Model (CMM) (Paulk et al. 1995), also tailored into SW-CMM (SW-CMM). The essence of this model is for organizations to strive for achieving higher maturity levels and thereby becoming better at software development. This model has also been tailored to the Test Maturity Model (TMM) (Veenendaal 2002) where focus is put on making the test process more efficient. Such models are focused on making an organization more structured and work according to what the model think is state of the art. However, the assumption that there is one state of the art model that fits all companies is flawed because there are no generally applicable solutions (Glass 2004). Further, such models do not evaluate the current improvement needs of an organization and in what context they are developing their software. For example, TMM can identify a missing process activity but it cannot tell how well existing activities work.

Another common approach to process improvement is to identify process flaws through fault analysis (Biehl 2004), (Chillarege et al. 1992), (Glass 2004). In fact, related research claims that fault analysis is the most promising approach to software process improvement (Grady 1992). Fault analysis has possibilities to achieve higher quality (Barrett et al. 1999), (Biehl 2004) or it might aim at decreasing fault removal costs during development (Chillarege et al. 1992), (Hevner 1997). One of the methods used in this chapter, 'Faults-Slip-Through' (FST), primarily aims at decreasing fault removal costs (see Chapter 3). As previously mentioned, FST is defined as whether a fault slipped through the phase where it should have been found. The definition of it is similar to measures used in related work, e.g. phase containment metrics where faults should be found in the same phase as they were introduced (Hevner 1997), and

goodness measures where faults should be found in the earliest possible phase (Berling and Thelin 2003). FST can also be used to measure an organization's improvement potential by calculating the difference between the cost of faults in different phases in relation to what the fault cost would have been if none of them would have had slipped through the phase where they were supposed to be found. For example, if a fault that should have been found during implementation instead was found during system test and it costs 20 hours more to find in system test, the improvement potential for such a fault is 20 hours.

Orthogonal Defect Classification (ODC) can as mentioned in the introduction provide process feedback through fault type and fault trigger classifications (Chillarege et al. 1992). ODC fault type classification can provide development feedback by categorizing faults into cause related categories such as assignment, checking, interface, and timing (Chillarege et al. 1992). An attempt to apply this classification at the studied Ericsson department was first made but it failed mostly due to the inability to make unambiguous classifications, i.e. it was not obvious which category a fault should belong to. Related research has also recognized this problem (Henningsson and Wohlin 2004). Further, the corrective actions to make from this kind of classification were not obvious, e.g. there could be several causes of having many assignment faults. On the contrary, ODC fault trigger classification appeared to be promising since test related activities seemed easier to separate from each other and could also more easily be connected to a certain improvement action. A fault trigger is a test activity that makes a fault surface and an ODC trigger scheme divides faults into categories such as concurrency, coverage, variation, workload, recovery, and configuration (Chillarege and Prasad 2002). Further, a trigger scheme is relatively easy to use since the cause of each fault needs not to be determined; only the testers' fault observations are needed. Commonly, ODC fault triggers are used for identifying discrepancies between the types of faults the customers find and those that are found during testing. Through such analysis, test improvements can be made to ensure that customers find fewer faults in consecutive releases (Barrett et al. 1999). Dividing trigger distributions after which phases the faults were found in can show what types of faults different phases find. However, when improving test efficiency, it is probably not the fault prone phases that need to be improved but rather an earlier phase that needs attention. The ODC fault trigger concept attempts to manage this by assigning each trigger class to a specific phase that should uncover that fault (Barrett et al. 1999). However, this implies that each trigger activity only can belong to one phase and in organizations having several test levels and several test activities, this results in a vast number of fault triggers that not only are difficult for the testers to learn but also become very process dependent. In fact, an attempt to adopt this approach at the studied department failed due to that reason. Thus, the method described in the next section intends to handle situations

when organizations request a more generic and simple trigger scheme and still wants to relate to fault belonging phase.

6.3 Method

This section describes the developed evaluation method and the case study environment where its applicability was demonstrated.

6.3.1 Combining Faults-Slip-Through and ODC Fault Triggers

Section 6.2 explained how FST can identify phases that need improvements and ODC fault triggers can identify which activities that need to be improved. This section describes how these methods can be combined to not only identify which phases that need to be improved to decrease FST but also which activities in each phase that are the sources of a high FST.

Table 6.1: ODC Fault Trigger Scheme

Fault trigger	**Description**
Coverage	Single, straightforward execution of a single function
Variation	Execution of a single function, but using combinations of parameters
Sequencing	Execution of multiple functions in a specific sequence
Interaction	Execution of multiple functions in complex combinations
Workload	Execution with a high workload
Recovery	Execution of error cases
Startup/restart	Executions that start and restart the system
Hardware configuration	Hardware configuration tests
Software configuration	Software configuration tests

The first step in achieving this objective is to construct a fault-trigger classification-scheme that has non-overlapping categories, is easy to use, and generic within the context of the organization. It is suggested that this scheme is obtained through iterative development, i.e. make a draft, test it on a set of faults and employees, then refine it until satisfactory. This is also how the scheme used in the case study in this paper was obtained. Table 6.1 illustrates the traditional ODC trigger scheme commonly used

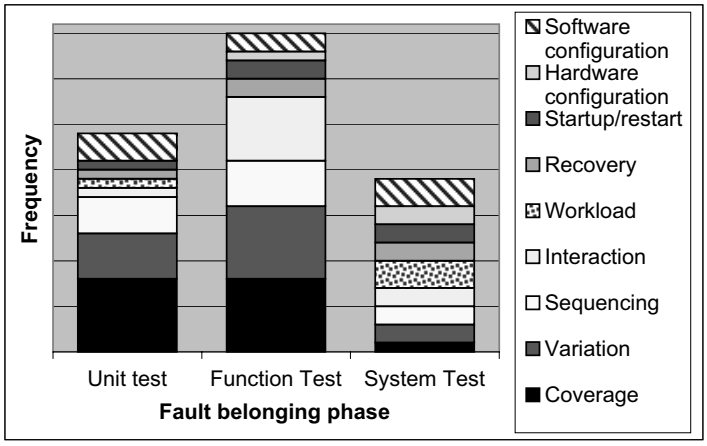


Figure 6.1: Example: Combination of Faults-Slip-Through and ODC Fault Triggers

in related research (Butcher et al. 2002). Besides tailoring this standard scheme because of contextual differences, having few faults to classify might make it feasible to decrease the amount of triggers categories, i.e. to obtain a higher statistical validity (Barrett et al. 1999).

From the obtained trigger scheme, all faults should be categorized after which trigger they belong to. That is, when having the faults already classified according to the previously described FST method, the next step is to sort the FST after which phase they should have been found in together with their trigger classification. This also means that the faults that are not classified as FST are excluded from the classification result. The result of such a classification is illustrated in the fictitious example in Figure 6.1. In the figure, one can for example see that the unit test phase has a quite high slippage of coverage and variation faults to later phases.

The fault distribution in Figure 6.1 provides useful information about which triggers that might need attention. However, a problem identified during practical application was that it might not always be the highest columns that cause the most problems in testing. For example, workload faults are likely to be more costly to correct than coverage faults and configuration faults has a higher test cost because they are more likely to be ‘stopping’, i.e. they cause re-installation during which further testing cannot be performed. Therefore, a cost variable should be added to the trigger distributions. Since the fault cost of each fault commonly is not accessible, an average fault cost matrix is

suggested, i.e. the average fault cost for each trigger type for each phase (faults cost more when found in later phases (Boehm 1983), (Shull et al. 2002)). Table 6.2 illustrates an example of such a cost estimation, e.g. a workload fault costs 15 hours to fix when found in System Test. Each value can be determined either by measuring the cost of each fault when fixed or by using expert judgements. Although expert judgements are not exactly accurate, they are fast to obtain and for this purpose normally accurate enough.

Table 6.2: Example of Fault Cost (Fault Trigger/Phase)

Fault trigger	**Unit test**	**Function Test**	**System Test**
Coverage	1	2	4
Variation	1	2	4
Sequencing	1	2	4
Interaction	1	2	4
Workload	5	10	15
Recovery	3	5	7
Startup/restart	2	4	8
Hardware configuration	3	6	10
Software configuration	3	6	10

When applying the cost estimates in Table 6.2 on each fault in Figure 6.1, the weighted trigger distribution is obtained as shown in Figure 6.2, sorted after which phase the faults belonged to. As can be seen in the figure, for example workload and configuration faults now have the largest proportions in System Test. Therefore, these should be prioritized when deciding upon improvement actions.

6.3.2 Case study

The practical applicability of the method described in the previous section was demonstrated through a case study at a software development department at Ericsson AB. The department runs several projects in parallel and consecutively. The projects develop new functionality to be included in new releases of existing products that are in full operation as parts of operators' mobile networks. A typical project such as the one studied in this paper lasts about 1-1.5 year and has on average about 50 participants. The products are verified in three steps: Basic Test, Function Test, and System Test. Since the products constitute a smaller part of large mobile networks, they are rather difficult to verify. Therefore, verification against other product nodes in the mobile

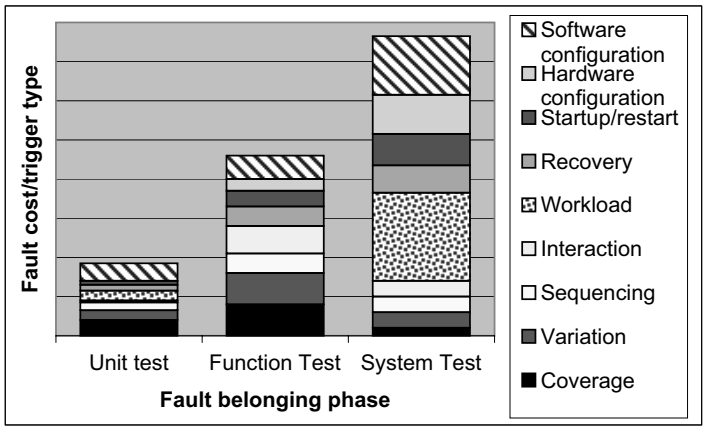


Figure 6.2: Example: Trigger Cost Distribution (FST)

networks is a large part of System Test. Before releasing the product globally, initial field tests are performed together with a trial customer.

In the studied project, only faults found as of Function Test were stored in a fault reporting system, i.e. faults found earlier were not reported in a written form that could be post-analyzed. Finally, some of the reported faults were excluded because they were false positives or they did not affect the operability of the products, e.g. opinion about function, not reproducible faults, and documentation faults.

6.4 Case Study Results

The purpose of this section is to demonstrate the applicability of the method suggested in this paper. This was achieved through a case study on a software development project at Ericsson AB. Section 6.4.1 describes the fault trigger scheme that was developed for the case study and then sections 6.4.2 and 6.4.3 present the results of applying the scheme together with FST, i.e. relative number of faults and relative fault cost respectively. Finally, Section 6.4.4 describes how the results were used in practise, e.g. in form of changes made in the test process at the Ericsson department.

6.4.1 Construction of Fault Trigger Scheme

In order to obtain a useful fault trigger classification, the applied fault trigger scheme needed to be mapped against the test process of the organization to study. Although, a standard ODC fault trigger scheme was good to have as a starting point, it needed modifications before being suitable to apply in the organization to study. In the case study, the scheme was obtained through trial test usage in different products at the department and then iteratively refined until satisfactory. Table 6.3 presents the resulting list of fault triggers together with descriptions of the scope of each trigger.

Table 6.3: Applied Fault Trigger Scheme

Fault trigger	Description
Internal interface	Fault in interface between components.
ExternalInt-Struct	Fault in external interface, e.g. protocol specification
ExternalInt-Data	Fault in data sent through external interface.
Human interface	Fault in GUI, i.e. in human interaction.
Performance	Insufficient performance.
Robustness	Load and endurance faults (including memory leaks)
Redundancy	Fault in data replication, recovery, cluster operation (except installation related faults).
Concurrency	Includes timing, usage of shared resources, i.e. faults that only occur at certain time-points or when affected
Configuration	Software and hardware configuration faults, e.g. installation faults. Also includes version faults.

6.4.2 Combination of Faults-Slip-Through and Fault Triggers

The trigger scheme described in the previous section was applied on a typical software development project at the Ericsson department. Analyzing the phase belonging, i.e. whether a fault was a FST, was previously also made on all faults in the studied project. Figure 6.3 describes the result of combining these classifications. The possible phases (i.e. columns) in the figure that each fault could belong to were as follows:

Basic Test (BT): Faults found during unit tests of a component.

Integration Test (IT): Faults found during primary component integration tests, e.g. installation faults and basic component interaction faults.

Function Test (FT): Faults found when testing the features of the system.

System Test (ST): Faults found when integrating with external systems and when testing non-functional requirements.

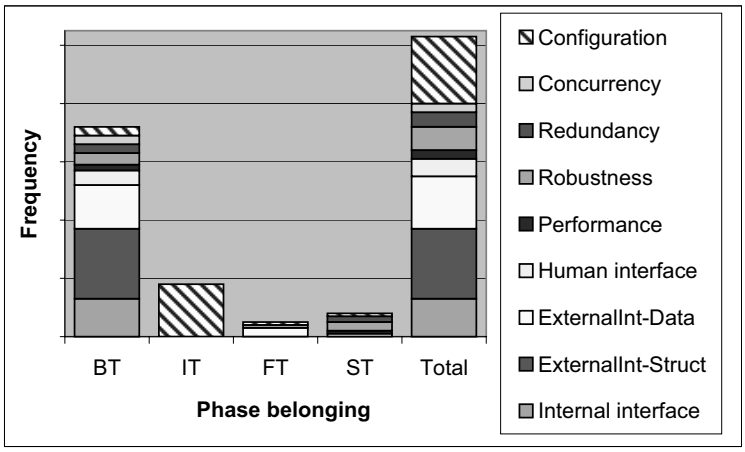


Figure 6.3: Combination of FST and Fault Triggers

As can be seen in the left column in Figure 6.3, a significant amount of faults in external interfaces slipped through the BT phase. Further, the IT phase had several configuration faults that slipped through to later phases whereas FT and ST only had smaller amounts of each trigger type that slipped through to later phases. The phase FiT+6 was not included in this figure since no faults should be found in that phase, i.e. all faults that were found during this phase were classified as slips from earlier phases. Therefore, the column would have been empty if included.

6.4.3 Fault Trigger Cost

As exemplified in the method description in Section 6.3.1, it is not only the amount of faults for each fault trigger that affects which types of faults that need attention. The cost difference between different fault trigger types might be significant and should therefore also be considered in the trigger analysis. Table 6.4 presents the relative cost of each fault trigger in relation to phase and trigger type as estimated through expert judgements at the Ericsson department. These estimations are therefore not exact but are good enough to later illustrate the trigger distributions with the cost aspect in mind. The rightmost column, FiT+6, corresponds to faults found during field tests with a trial customer.

When applying the fault costs in Table 6.4 on the fault trigger distribution in Figure

Table 6.4: Average Cost of Each Fault Trigger

Cost of faults	BT	IT	FT	ST	FiT+6
Internal interface	1	2	5	10	15
External interface - Struct	1	2	5	10	15
External interface - Data	1	2	5	10	15
Human interface	1	2	5	10	15
Performance	10	10	20	35	50
Robustness	15	15	25	45	60
Redundancy	10	10	20	30	40
Concurrency	10	10	20	30	40
Configuration	10	10	20	30	40

6.3, a new cost-weighted trigger distribution could be obtained. Figure 6.4 illustrates the result of this calculation and as can be seen in the figure, the importance of the different triggers became slightly different. For example, since robustness faults cost relatively more to find and fix, that category now also has a significant proportion in all phases. The cost of configuration faults could also be decreased, especially in the IT phase.

6.4.4 Performed Improvement Actions from the Case Study Results

The results presented in the previous two sections were used as input to a workshop where areas to improve should be identified and prioritized. From the workshop discussions, external interface tests and robustness tests in the BT phase, and configuration tests in the IT phase were the primary areas that were selected to be improved in the next subsequent project. For each improvement area, a follow-up workshop was held to identify specific process changes that would increase the ability of that activity to find more faults.

6.5 Discussion

This section lists some lessons learned when applying this method followed by a discussion on validity threats to the results.

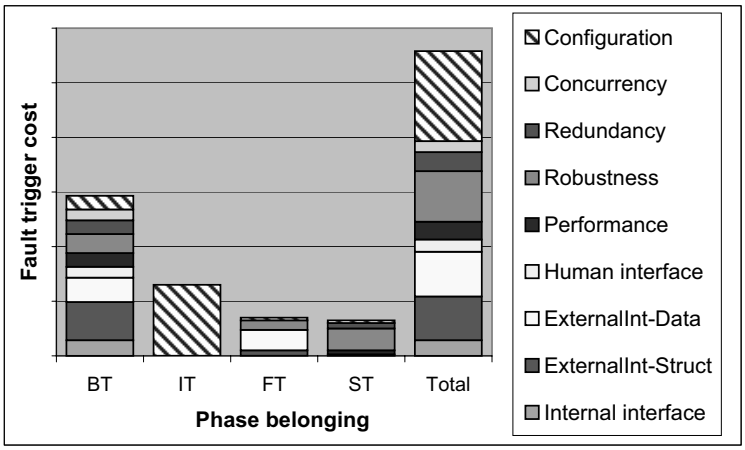


Figure 6.4: Trigger Cost Distribution (FST)

6.5.1 Lessons Learned

The major lesson learned is that the proposed classification method appeared to be useful in practice. However, measurements are not supposed to replace human judgements. Metrics are good for guidance, not absolute recipes (Voas 1999). Nevertheless, at the studied department, the obtained metrics data was considered as an important input to stimulating discussions and improvements. Still, an organization should not necessarily rely on one single measure. For example, total number of faults found in different phases and identification of fault prone components might also provide valuable input to improvement actions.

Regarding creation of the trigger classification scheme, the recommendation is as earlier stated to tailor it to the specific context. However, it should still be general within the context of organization so that projects and products can be compared against each other. For example, the distributions can determine how products differ in terms of test challenges. This is for example important when striving for using common test processes in an organization developing several products.

Finally, in order to be able to ensure that an improvement has been implemented successfully, managers should also follow up the improvement quantitatively. This can naturally be achieved by comparing certain columns in the trigger distribution before and after the improvement. Preferably, a goal value for each improved trigger type

should also be set and monitored. Existing techniques for goal follow-up such as the Goal-Question-Metric (GQM) paradigm (Basili and Green 1994) could be used as support for this part.

6.5.2 Validity Threats

When conducting an empirical industry study, the environment cannot be controlled to the same extent as in isolated research experiments. In order to be able to make a correct interpretation of the results presented in this paper, one should be aware of threats to the validity of them. As presented below, the main validity threats to the performed case study concern conclusion, internal, and external validity (Wohlin et al. 2003). Construct validity is not relevant in this context since the case study was conducted in an industrial setting.

Conclusion validity concerns whether it is possible to draw correct conclusions from the results, e.g. reliability of the results (Wohlin et al. 2003). The threats to conclusion validity are as follows.

In order to be able to draw conclusions from the results, the department must have a common view on which phase each fault should belong to. That is, managers and developers should together agree on which fault types that should be considered as FST and not. At the studied department, this was managed by having workshops where checklists for how to estimate FST were developed. Further, regarding the average fault cost for different triggers in different phases, the result was obtained through expert judgments and therefore, the estimations might not exactly reflect the reality. However, the estimations were considered good enough to serve the objective, i.e. input to Figure 6.4. However, when possible, direct fault cost measures should be included in the fault reports so that this uncertainty is removed.

Internal validity concerns how well the study design allows the researchers to draw conclusions from causes and effects, i.e. causality. For example, there might be factors that affect the dependent variables (e.g. fault distributions) without the researchers knowing about it (Wohlin et al. 2003). In the case study presented in this paper all faults were post-classified by one researcher, which thereby minimized the risk for biased or inconsistent classifications. Another threat to internal validity is whether certain events that occurred during the studied projects affected the fault distribution, i.e. events that the researchers were not aware of. This was managed through workshops with project participants where possible threats to the validity of the results were put forward.

External validity concerns whether the results are generalizable or not (Wohlin et al. 2003). In this case study, the results are in overall only generalizable within the context of the case study. Nevertheless, in this paper, the main concern regarding

external validity is whether the method used for obtaining the results is generalizable or not. Given that the trigger scheme and average trigger cost data are adjusted in each context to be applied, the method is fully applicable in other contexts. Thus, the method is externally valid since it can be replicated in other environments.

6.6 Conclusions

The purpose of this paper was to determine how fault analysis can be used to identify which activities in an organization's test process that need to be improved. Specifically, the hypothesis was that two existing fault analysis methods: ODC fault trigger classification and Faults-Slip-Through (FST) could be combined to provide useful input to test process improvement. The method described in this paper achieved this by classifying faults both after fault trigger categories and after which phase they should be found in. That is, it is in a fault distribution graph possible to see which activities that need to be improved and in which test phase the improvement for each activity is needed.

The practical applicability of the proposed method was demonstrated by applying it on an industrial software development project at Ericsson AB. Through the output provided by the method, areas to improve in the next consecutive project were prioritized.

Chapter 7

Monitoring Test Process Improvement

Lars-Ola Damm and Lars Lundberg

Abstract

In a highly competitive environment where time-to-market is crucial for success, software development companies initiate process improvement programs that can reduce the development cost. They especially seek improvements in the test phases since rework commonly constitutes a significant part of the development cost. However, the success of process improvement initiatives is dependent on early and observable results since a lack of feedback on the effect of improvements is a common cause of failure. This paper presents a method that uses a fault-based approach to monitor improvements. The method was applied in two case study projects at Ericsson AB, one fully conducted at a main site and one partly conducted at an offshore location. Through the application of the method, managers could monitor the test process status of the projects and use the output as decision support to resolve issues early. Additionally, several insights were gotten from the case studies, e.g. cultural differences affect how companies should approach metrics programs in global software development.

7.1 Introduction

Avoidable rework is commonly a large part of a development project, i.e. 20-80 percent depending on the maturity of the organization and the type of development projects (Shull et al. 2002), (Veenendaal 2002). Therefore, software development companies are interested in process improvement programs that can reduce the rework cost. There are several ways to identify causes of high rework costs, e.g. fault based approaches (Leszak et al. 2000), (Chillarege and Prasad 2002). However, many improvement efforts fail because turning assessment results into actions is difficult to achieve (Mathiassen et al. 2002). In fact, the failure rate of process improvement implementation is reported to be about 70 percent (Ngwenyama and Nielsen 2003). Therefore, practitioners request more guidance on how, not just what to improve (Rainer and Hall 2002), (Niazi et al. 2005).

According to related research, a lack of attention to feedback in the process is a major cause of the failure of several technical innovations and improvement paradigms (Gray and Smith 1998). The success of a process improvement initiative is dependent on early and observable results (Mathiassen et al. 2002). This is especially important in offshore development, i.e. in globally distributed development where long distances and cultural differences make implementations of improvements even harder to monitor.

A department at Ericsson develops some software products on its main site and one partly at an offshore location. The department wanted to be able to monitor ongoing test process improvements in these projects, e.g. improvements that should ensure that each test phase covered what they should. The department especially requested ways to monitor the offshore-developed product since large distances make it harder to control. Further, they experienced that it is easy to tell an offshore department to follow a certain test process but cultural differences and geographical distances make it hard to know if the department really adheres to the process. They might just claim that they adhere to the process to please management or think they do but have misunderstood fundamental parts of it. Therefore, the department managers requested an easy method for monitoring process adherence. In this context, easy implied that straightforward measurements should automatically monitor the status so that eventual issues would surface directly. That is, a more time-consuming post-mortem analysis or process audit was not feasible for this purpose.

Instead, another possible solution was identified, i.e. as described in Chapter 5, previous practical work in analysis of fault distributions had shown potential regarding possibilities to evaluate the test process that a development organization uses. The basic measure of this concept estimates the amount of faults that should have been found in an earlier phase, i.e. 'Faults-Slip-Through' (FST). That is, the faults are classified

according to whether they slipped through the phase where they should have been found. The basic motivation for the concept is based on the commonly accepted fact that faults in overall are cheaper to find earlier (Boehm 1983), (Shull et al. 2002). After a completed project, a post-analysis of the number of faults that were slips and not should be counted to determine how good the project was at identifying faults in the right phase. A definition of which faults should be found in which phase, i.e. the test strategy, serves as input to the estimations. Thus, the FST post-analysis determines the degree of fitness to a specified test strategy. This implies that the measure is indirect in that it ensures that the goals of the process are fulfilled rather than checking if specific process steps are conducted. From this measurement possibility, the main research question of this paper is as follows:

How can faults-slip-through (FST) be used to guide and monitor software test process improvement?

To address this research question, this paper presents a method and the results from practical application in two case study projects: one developed on-site and one partly developed at an offshore location.

The paper is outlined as follows. Section 7.2 provides an overview of related work in this research area. Section 7.3 describes the method and the setting of the case study where the method was applied. The results from the case study are provided in Section 7.4. Section 7.5 discusses the results and eventual validity issues and finally, Section 7.6 concludes the work.

7.2 Related Work

7.2.1 Test Process Improvement

Within software process improvement, one of the most widely used models is the Capability Maturity Model (CMM) (Paulk et al. 1995). The essence of this model and its tailored variant SW-CMM (SW-CMM) is for organizations to strive for achieving higher maturity levels and thereby become better at software development. This model has also been tailored to the Test Maturity Model (TMM) (Veenendaal 2002) where focus is put on making the test process more efficient. Another related area to such maturity models is the notion of quality certification, i.e. traditionally through the ISO standard (ISO 1991), now enhanced into ISO SPICE (El Emam et al. 1998). However, such maturity and assessment models have been criticized because they are focused on making an organization more structured and work according to what the model think is state of the art. That is, although state of the art models can provide some guidance to

areas of improvement, there are no generally applicable solutions (Glass 2004), (Mathiassen et al. 2002). The models should align with company practices, not industry benchmarks that are not tailored to the context (Glass 2004), (Mathiassen et al. 2002). Further, since the software process is a creative, feedback driven, learning and adaptation process, such concepts are of particular importance for process improvement. However, best practice based improvement paradigms such as CMM do not perceive that (Gray and Smith 1998).

The opposite of applying a model-based approach is the bottom-up approach where improvements are identified and implemented locally in a problem-based fashion (Jakobsen 1998). The basic motivation for using the bottom-approach instead of the top-down approach is that process improvements should focus on problems in the current process instead of trying to follow what some consider to be best practices (Beecham and Hall 2003), (Glass 2004), (Jakobsen 1998), (Mathiassen et al. 2002). That is, just because a technique works well in one context does not mean that it also will in another (Glass 2004).

A typical bottom-up approach is the Quality Improvement Paradigm (QIP), where a six-step improvement cycle guides an organization through continuous improvements based on QIP (Basili and Green 1994). From defined problem based improvements, QIP sets measurable goals to follow-up after the implementation. The method implemented in this paper can be considered conceptually similar to QIP but tailored to be a more concrete realization aimed at how to continuously monitor test process improvements. That is, QIP is more of a generic framework for which steps to include in an improvement cycle, it does not state exactly how to perform them.

Within the area of monitoring and feedback on process improvement implementations, some research has been made. A typical example is the area of Statistical Process Control (SPC) that originally was developed for monitoring quality control of production lines but also have been applied on the software development process (Cangussu et al. 2003). Within test process improvement, a typical area of interest has been the number of faults found and remaining in a product, e.g. Six Sigma which focus on decreasing the number of faults found per produced product unit (SixSigma 2005). Further, other fault classification techniques have been specifically tailored to monitor test efficiency of the test process already during development (Bhandari et al. 1993). The idea is that by monitoring which types of faults that occur frequently directly during ongoing testing, problem areas can be identified and resolved (Bhandari et al. 1993). However, such techniques only identify fault types that occur frequently; they do not attempt to assess the overall quality of the product.

The classification method used in this paper, ‘Faults-Slip-Through’ (FST), was primarily developed for identifying opportunities for decreased fault costs (see Chapter 3). As mentioned in the introduction, FST is defined as whether a fault slipped through

the phase where it should have been found. The definition of it is similar to measures used in related work, e.g. phase containment metrics where faults should be found in the same phase as they were introduced (Hevner 1997), and goodness measures where faults should be found in the earliest possible phase (Berling and Thelin 2003). Thus, the primary difference is that when defining FST, the test efficiency is in focus; the fault cause is not considered.

The FST method has as described in Chapter 5 previously been successfully applied for post-evaluation of an improvement aimed at finding more faults earlier. However, no monitoring support was provided in this approach since the evaluation was only performed upon project completion.

7.2.2 Global Software Development

Since one of the studied projects in this paper was partly developed at an offshore location, the notion of global software development should also be mentioned here. First of all, it becomes more and more common to outsource parts of software development efforts to offshore locations where costs are low and labour often is plentiful (Krishna et al. 2004). There are two main options for how to approach such distributed development, i.e. offshore outsourcing through contracting services with an external organization or offshore insourcing where the subsidiary is owned by the procurer (Prikladnicki et al. 2003). The offshore department studied in this paper belongs to the category offshore insourcing.

Besides the need to educate the offshore department on the techniques and processes to use, the cultural challenges affect the collaboration procedures. Hofstede has identified four key dimensions of culture that affect how people behave in relation to software process improvement (Hofstede 1996):

Power distance: describes the extent to which the decision power of a company is hierarchical or distributed. For example, people in cultures with a large power distance strictly work according to what their managers instruct them to do.

Uncertainty avoidance: indicates the extent to the employees prefer strict rules to avoid ambiguous situations. Such people also tend to refuse to tolerate deviance from defined rules.

Masculinity versus femininity: determines if a society to prefer masculine aspects such as competitiveness and materialism or feminism aspects such as relationships, and social well being.

Individualism versus collectivism: indicates if people tend to be independent or prefer collective interdependence. That is, do people focus on what is best for themselves or are they team players?

Although there are large differences between different nations, few cultures are fully at any of the extremes (Hofstede 1996). Together with the impact of geographical distance, typical implications of cultural differences are that processes need to be well-specified, e.g. have clear entry exit criteria (Prikladnicki et al. 2003). Some companies manages the cultural differences by outsourcing to a company that has a similar mindset (Krishna et al. 2004). For example, Norwegians tend to outsource to Russians instead of Asians (Krishna et al. 2004).

7.3 Method

This section describes the suggested method for addressing the stated research question and the context of the case study where the method was applied.

7.3.1 FST Definition and Follow-up Process

The selected method for guiding and monitoring test process improvement was primarily obtained through practical experience from pilot tests on an on-site conducted project. Nevertheless, as described in the introduction, the method was considered especially useful in offshore-developed projects and was therefore also adapted to fit that purpose.

1. Define how to work, i.e. specify the test strategy

As described in Chapter 3, the FST definition is created by specifying which types of faults that should be found in each phase. For example, a criterion for the unit test phase could be that all memory leaks should bee found there. In this paper, only test phases were considered although earlier development phases also could be included. The reason for excluding earlier phases was that the case study department preferred to have a focus on test strategies. To ensure that the definition would be understood and used correctly, it is important both to involve developers and testers in the creation of it. High stakeholder involvement in process improvement initiatives is in related work also recognized as a key factor for success (Mathiassen et al. 2002), (Rainer and Hall 2003).

2. Determine the current FST baseline level

To be able to evaluate the effect of an improvement, a baseline value must first be obtained. This is achieved by classifying the faults of a baseline project according to the FST definition. This could be done as a post-mortem analysis. Table 1 describes an example of the result of such an analysis. In the example, 20 faults were found in the Function Test phase of which 5 were slippages from earlier phases. This results in an accumulated result of 25 percent FST to Function Test. In our experience, evaluations

should prefer to use the percentage value instead of using number of faults because the numbers of faults are directly dependent on the project size. Further, this measure could also be applied on faults found during the maintenance phase. However, in this case, the measure foremost assesses the obtained quality of the product rather than the efficiency of the test phases.

Table 7.1: Example of FST Measurement

	No. faults found	No. FST	% FST
Function Test	20	5	5/20=25%
System Test	8	4	4/8=50%

3. Set improvement goals for the next-coming project

The improvement goals should be based on the baseline values of the previous project. Depending on the improvement actions to be taken, a suitable level of percentage improvement should be set as a goal, i.e. X percent better than the previous project. Selecting a suitable improvement level is rather much a matter of experience. However, if for example extending this measure with a root cause analysis and consecutive actions to prevent certain set of faults, it is from that possible to determine a likely improvement level.

Further, the reason for not just specifying 'hard' numbers to reach is that it foremost is the striving for improvement that is important, not the underlying numbers. As further discussed in Section 7.5.2, there are also other positive implications of using degree of improvement instead of the underlying numbers.

4. Continuous goal follow-up during project execution

To make sure that progress is made, one need to follow-up goals regularly to ensure that the project really strive to achieve the goals. This should be done regularly during the project and not just afterwards because then it is too late to adjust eventual issues. However, this sets requirements on the fault reporting system. That is, the faults most be classified directly when solving the fault reports and a tool for extracting the statistics automatically is preferable. For example, if the fault reports are stored in a database, the tool should fetch the reported FST value of each fault report and then aggregate and calculate the result. 7.3.2 describes an example of how to measure the status. In the example, Release R of a product had 25 percent FST to Function Test whereas the status of Release R+1 is 20 percent. This gives an improvement status of 20 percent. However, in the System Test example, the status is a decrease of 10 percent, which indicates a problem to be addressed by the project. The case study results in Section 7.4 further exemplify how the status can be continuously monitored.

Table 7.2: Example of FST Follow-up

FST status	Release R	Release R+1	Impr. status
To Function Test	25%	20%	(25-20)/25=+20%
To System Test	50%	55%	(50-55)/50=-10%

Further, the status should be made visible to everyone in the monitored project because people tend to work according to how they are measured (Rakitin 2001). Visibility could for example be obtained through weekly or monthly progress reports, or through regular status meetings. Having the statistics generated onto a webpage further increases the visibility.

7.3.2 Case Study Setting

The practical applicability of the method described in the previous section was demonstrated through two case studies at a software development department at Ericsson AB. The department runs several projects in parallel and consecutively. The projects develop new functionality to be included in new releases of existing products that are in full operation as parts of operators' mobile networks. A typical project such as the ones studied in this paper lasts about 1-1.5 year and has on average about 50 participants.

The products are verified in three steps: Basic Test, Function Test, and System Test. Since the products constitute a smaller part of large mobile networks, they are rather difficult to verify. Therefore, verification against other product nodes in the mobile networks is a large part of System Test.

One of the studied projects was partly developed at an offshore located software development department. This offshore department completes the phases from low-level design to Function Test. That is, the main design center handles the phases before and after that including System Design and System Test.

In the studied projects, only faults found in Function Test and System Test were stored in a fault reporting system, i.e. faults found earlier were not reported in a written form that could be post-analyzed. Therefore, only slippages to Function Test and System Test were measured. Finally, some of the reported faults were excluded because they did not affect the operability of the products, e.g. opinion about function, not reproducible faults, mistakenly reported faults, and documentation faults. Low priority faults such as cosmetic faults were also excluded.

Regarding the products of the two studied projects, the differences were quite small. That is, both products had existed on the market for a while and were developed on the

same platform. The only relevant difference was that the offshore-developed product had more external protocol interfaces but lower performance and robustness requirements.

7.4 Results

This section describes the results from applying the method described in the previous section in two software development projects of which one was developed at an offshore department.

1. Define how to work, i.e. specify the test strategy

The studied organization first developed an FST definition for two on-site developed products. The definition was obtained through workshops where key representatives from different areas at the department participated. The output from the workshop was a checklist specifying which types of faults that should belong to which phase.

With this as basis, a tailored definition for the offshore department was also developed through a set of workshops with relevant stakeholders involved, i.e. managers, developers, and testers from both development sites. Appendix C describes the obtained definition.

2. Determine the current FST baseline level

From the obtained FST definition, a post-analysis of the predecessor projects was made, i.e. to obtain baseline values. In the on-site developed baseline project, the FST was 71 percent to Function Test and 55 percent to System Test. The offshore-developed project had an FST of 39 percent to Function Test and 32 percent to System Test. Due to confidentiality reasons, the actual number of faults are not presented here. The remarkable part of these results was that the offshore project had lower FST values than the on-site developed project. That is, the results were surprising because the offshore product had a history of more customer faults than the on-site product and was also considered as less mature. Section 7.5 further discusses the causes and impact of this difference.

3. Set improvement goals for the next-coming project

In the on-site developed project, the improvement goals for the consecutive release were set to ten percent improvement for both Function Test and System Test.

For the offshore-developed project, the corresponding values were a five percent improvement for both Function Test and System. The reason for not setting higher goals was that striving for close to zero slippage rates would result in significantly increased test efforts that would decrease test efficiency instead. That is, striving for 100 percent test coverage was not the aim. Experiences from evaluations of previous

projects also indicated that the specified improvement levels seemed moderately hard to achieve. Section 7.5 further discusses experiences from how to set the goals.

4. Continuous goal follow-up during project execution

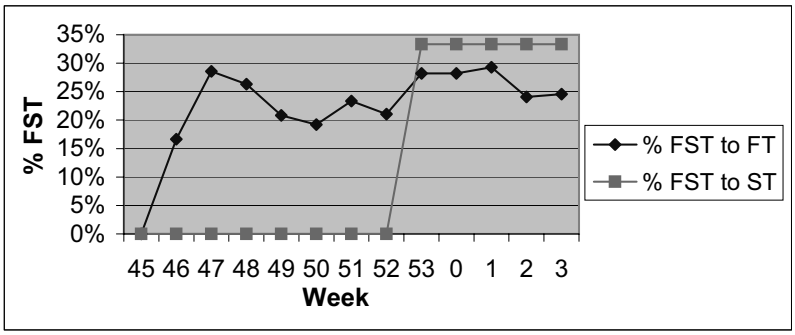
The result monitoring of FST for the studied projects were centred around a monthly status meeting, steered by the manager of design, coding, and test of all products in the organization. The primary input to the meetings was a webpage containing the current FST data for the currently running projects. This webpage was automatically updated on a daily basis by downloading fault report data from the projects, i.e. the FST measure was included in all fault reports and could be fetched by the web application. The web application presented the aggregated data in form of for example number of faults, number of FST and the current improvement status in relation to the previous project.

In the on-site developed baseline project, the aggregated FST upon completion was 63 percent to Function Test and 66 percent to System Test. That is, in comparison to the baseline projects, the FST rate to Function Test improved by 11 percent whereas the FST rate to System Test turned out to be 19 percent worse. The unwanted System Test results were indicated already in the first measures obtained the first weeks after the test phase started. It was concluded that the introduction of a new test automation tool in Function Test had caused too much focus to be put on the test areas covered by the tool whereas others were not adequately covered. However, this was adjusted in consecutive increments and during the rest of the test period, the values steadily improved although they never reached the same goal level.

During the design and implementation phases of the offshore developed project, the monthly meetings focused on making sure that the measure would be reported correctly when Function Test started. That is, it was made sure that the FST definition was agreed on and that the project members were educated. The monthly meetings and the web application turned out to affect the offshore site significantly since they through the attention realized the visibility of the measurement results and thereby that they should do their best to improve according to it.

Figure 1 presents the FST measurement results of the offshore-developed project, which at the completion of this paper still was ongoing. The figure presents the FST trend during Function Test and System Test. Regarding FST to FT, the percent FST was initially rather high but later stabilized to a more than satisfying level. In fact, the last data point indicates a FST to FT of 25 percent, which results in an improvement of 36 percent in comparison to the baseline value of 39 percent, i.e. (39-25)/39 = 36 percent. Regarding System Test, the phase did not start until week 52. Therefore, the slippage rate was as can be seen in the figure zero until then. The last data point of 33 percent is almost the same as the baseline level of 32 percent. However, at such an early stage of the test phase, the value is not very alarming since previous experiences have indicated that the slippage rates are higher in the beginning of each test phase.

Figure 7.1: FST Trend in the Offshore Project



7.5 Discussion

This section discusses the obtained results, i.e. how they should be interpreted. After that, a few lessons learned from the case studies are presented. Finally, validity threats and some other considerations are discussed.

7.5.1 Result Analysis

As reported in the result section, the case study projects had unexpected differences in the results, i.e. the offshore developed project had a lower FST. The identified causes of these differences were the following:

- Regarding FST to FT, the agreed offshore FST definition was not as strict as the on-site definition. That is, differences in the product architecture made the on-site product require more thorough testing of isolated components before FT whereas the offshore site considered some fault types to belong to Function Test no matter how trivial they were. The effect of this was that such faults were not classified as slips although they occurred frequently.
- The main reason for a low FST to System Test was that the offshore project conducted lesser parallel testing, which can affect the FST results negatively. That is, in the studied organization, a commonly applied approach for cutting lead-time is by applying parallel testing, e.g. start System Test before the Function Test phase is completed. However, parallel testing increases the risk for fault slippages since the tested code commonly has dependencies to untested code.

Therefore, it is difficult to direct such premature System Tests to only System Test the Function Tested code.

- Cultural differences affected both FST to FT and ST. The reason for this was that FST is a subjective measure, i.e. although a good definition can set constraints, there are always border cases. When making a post validation of reported statistics, one could see a clear distinction in how the developers chose fault belonging. That is, on-site developers tended to be self-critical, i.e. in the border cases, they often set those faults as slips. On the other hand, the offshore developers commonly set the border cases not to be fault slippages.

7.5.2 Lessons Learned

1. Developing the definition together with the offshore department increased their understanding of and commitment to it. Further, this also increased the understanding of the current way of working at the offshore department.

2. The obtained FST measure must be put in context, i.e. how difficult the defined test strategy is to adhere to affects how low FST that can be obtained. For example, this was observed in the first bullet in Section 7.5.1. Therefore, FST should not be used to benchmark organizations against each other.

3. The FST measure does not only indicate when too many faults slip through a phase, sometimes a phase might also perform too much testing. That is, the right thing should be tested in the right phase; especially redundant work should be avoided. This implies that when setting goal values, the feasible improvement goal might differ between phases. For example, an improvement of ten percent to Function Test might be good whereas the System Test phase should have a zero improvement rate and instead consider how the current tests can be performed faster without increasing the fault slippage rates, e.g. through parallel testing as described in the previous section.

4. The case studies showed that cultural differences affect how measurements are perceived. The quite commonly accepted fact ‘People work according to how they are measured’ (Rakitin 2001) got a new meaning in the case studies, i.e. the statement is much more true in cultures having a large power distance (see Section 7.2.2). Cultures that do not have a large power distance accept the current situation and then improve on it whereas cultures with a large power distance tend to withhold accurate data if it is not in accordance with defined goals. The major underlying reason for this is that cultures with a large power distance tend

to have a lot more reward systems tied to their measurements. This was also the case in the offshore case study project. Such effects of metrics based reward systems have also been experienced in related research (McQuaid and Dekkers 2004).

5. A fault report 'guard' that rejects insufficiently filled in fault reports is recommendable to have at least in the beginning after introducing such a measure, e.g. to make sure that the measure is not forgotten or wrongly reported when answering the fault report. In multiple projects, this guard should preferably be the same person so that eventual discrepancies can be observed.

6. Automated tool support such as the web-based application used in the case studies in this paper has a strong positive influence on metrics success. This has also been recognized in other metrics research, i.e. manual overhead easily causes reported metrics not to be used (Gopal et al. 2002).

7.5.3 Validity Threats to the Results

The main validity threats to the results of the performed case study concern conclusion, internal, and external validity (Wohlin et al. 2003). Construct validity is not relevant in this context since the case study was conducted in an industrial setting.

Conclusion validity concerns whether it is possible to draw correct conclusions from the results, e.g. reliability of the results (Wohlin et al. 2003). The threats to conclusion validity are related to the lessons learned in the previous section, i.e., subjectiveness and cultural differences that affected the measurement outcomes. The approach used to address this was through pair-wise validations of the reported measures. That is, although for example cultural differences had an unavoidable impact, individual differences could at least be avoided by having the tester that filed the report to validate the FST value stated by the developer that entered it when answering the fault report.

Internal validity concerns how well the study design allows the researchers to draw conclusions from causes and effects, i.e. causality. For example, there might be factors that affect the dependent variables (e.g. fault distributions) without the researchers knowing about it (Wohlin et al. 2003). This was in overall not an issue because the primary purpose of this paper was to show how to obtain the results, not to measure the result of a certain treatment. This was therefore only an issue regarding the lessons learned, i.e. that the causes of the results were correctly interpreted. However, the risk for validity threats here was little because the lessons learned were not only observed by the researcher but also by the project staff. Further, previous experiences reported in the literature could in some cases also validate the lessons learned.

External validity concerns whether the results are generalizable or not (Wohlin et al. 2003). In this case study, the obtained results are in overall not generalizable since they are only valid for the studied projects. Nevertheless, the purpose of the case study was to demonstrate to applicability of the method, not to provide generalizable case study results.

7.6 Conclusions and Further Work

This paper presents a method for how a measure called Faults-Slip-Through (FST) can be used for guiding and monitoring test process improvements in software development. The method consists of the following steps:

1. Define how to work, i.e. specify the test strategy
2. Determine the current FST baseline level
3. Set improvement goals for the next-coming project
4. Continuous goal follow-up during project execution

The method was applied in two projects, of which one was developed at an offshore location. The case studies demonstrated the applicability of the method as it provided early feedback on the test process performance in the projects. Such feedback was valuable as decision support when determining which improvements to implement to achieve a more efficient testing or to improve the product quality. Further, a number of lessons learned were obtained including cultural differences that caused differences in the results of the two studied projects. Another lesson learned was that having automated tool support was a significant success factor for obtaining continuous measurement results.

Further work includes to introduce supporting methods that can provide quantitative support for resolving issues, e.g. by combining FST with other fault classification methods that can identify causes of an unexpected FST status in a certain phase. The impact of cultural differences on how to manage measurement programs also needs further investigations.

References

Aaen, I. (2003). Software Process Improvement: Blueprints Versus Recipes. *IEEE Software 20*(5), 86–93.

Ambler, S. (2004). *The Object Primer 3rd Edition Agile Model Driven Development with UML*. Cambridge University Press, Cambridge, UK.

Amland, S. (2000). Risk-Based Testing: Risk Analysis Fundamentals and Metrics for Software Testing Including a Financial Application Study. *The Journal of Systems and Software 53*(3), 287–295.

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

Baddoo, N. and T. Hall (2003). De-motivators for Software Process Improvement: An Analysis of Practitioners' Views. *Journal of Systems and Software 66*(1), 23–33.

Barrett, N., S. Martin, and C. Dislis (1999). Test Process Optimization: Closing the Gap in the Defect Spectrum. In *Proceedings of the International Test Conference*, pp. 124–129. IEEE.

Basili, V. (1992). Software Modelling and Measurement: The Goal Question Metric Paradigm, Computer Science Technical Report Series, CS-TR-2956 (UMIACS-TR-92-96). Technical report, University of Maryland, College Park, Maryland.

Basili, V. and S. Green (1994). Software Process Evolution at the SEL. *IEEE Software 11*(4), 58–67.

Bassin, K., T. Kratschmer, and P. Santhanam (1998). Evaluating Software Objectively. *IEEE Software 15*(6), 66–74.

Beck, K. (2001). 'Aim, Fire [test-first coding]'. *IEEE Software 18*(5), 87–89.

Beck, K. (2003). *Test Driven Development – by example*. Addison Wesley, Boston, MA.

Beecham, S. and T. Hall (2003). Software Process Improvement Problems in Twelve Software Companies: An Empirical Analysis. *Empirical Software Engineering 8*(1), 7–42.

Beizer, B. (1983). *Software Testing Techniques* (First ed.). Van Nostrand Reinhold Company, New York, NY.

Beizer, B. (1990). *Software Testing Techniques* (Second ed.). Van Nostrand Reinhold Company, New York, NY.

Berling, T. and T. Thelin (2003). An Industrial Case Study of the Verification and Validation Activities. In *Proceedings of the Ninth International Software Metrics Symposium*, pp. 226–238. IEEE.

Berwick, M. (1993). Optimising the Development and Test Process. In M. Kelly (Ed.), *Management and Measurement of Software Quality*, UNICOM SEMINARS, pp. 51–64. Unicom Applied Information Technology.

Bhandari, I., M. Halliday, E. Tarver, D. Brown, J. Chaar, and R. Chillarege (1993). A Case Study of Software Process Improvement During Development. *IEEE Transactions on Software Engineering 19*(12), 1157–1171.

Biehl, R. (2004). Six Sigma for Software. *IEEE Software 21*(2), 68–71.

Boehm, B. (1987). Improving Software Productivity. *Computer 20*(9), 43–57.

Boehm, B. and V. Basili (2001). Software Defect Reduction Top 10 List. *Computer 34*(1), 135–138.

Boehm, B. W. (1983). *Software Engineering Economics*. Prentice-Hall, Upper Saddle River, NJ.

Borjesson, A. and L. Mathiassen (2004). Successful Process Implementation. *IEEE Software 21*(4), 36–45.

Brooks, F. (1974). *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Publishing Company, MA.

Bundell, G., G. Lee, J. Morris, K. Parker, and L. Peng (2000). A Software Component Verification Tool. In *Proceedings of the International Conference on Software Methods and Tools*, pp. 137–146. IEEE Computer Society.

Butcher, M., H. Munro, and T. Kratschmer (2002). Improving Software Testing via ODC: Three Case Studies. *IBM Systems Journal 41*(1), 31–44.

Cangussu, J. W., R. A. DeCarlo, and A. P. Mathur (2003). Monitoring the Software Test Process using Statistical Process Control: A Logarithmic Approach. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 158 – 167. ACM SIGSOFT Software Engineering Notes.

Card, D. (1993). BOOTSTRAP: EUROPE'S ASSESSMENT METHOD. *IEEE Software 10*(3), 93–96.

Chillarege, R., I. Bhandari, D. H. M. Chaar, B. Moebus, B. Ray, and M. Wong (1992). Orthogonal Defect Classification—A Concept for In-Process Measurement. *IEEE Transactions on Software Engineering 18*(11), 943–956.

Chillarege, R. and K. Prasad (2002). Test and Development Process Retrospective - A Case Study Using ODC Triggers. In *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 669–678. IEEE Comput. Soc.

CMMI (2002). Capability Maturity Model Integration (CMMI), Version 1.1. CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMM-SE/SW/IPPD/SS, V1.1).

Cockburn, A. (2002). *Agile Software Development*. Addison-Wesley.

Conradi, R. and A. Fuggetta (July/August 2002). Improving Software Process Improvement. *IEEE Software 19*(4), 92–100.

Cook, E., L. Votta, and L. Wolf (1998). Cost-Effective Analysis of In-Place Software Processes. *IEEE Transactions on Software Engineering 24*(8), 650–662.

Creswell, J. W. (2003). *Research Design - Qualitative, Quantitative, and Mixed Methods Approaches* (Second Edition ed.). Sage Publications.

Damm, L.-O. (2002, June). Evaluating and Improving Test Efficiency, MSE-2002-15. Master's thesis.

DeMarco, T. (1997). *The Deadline- A Novel about Project Management*. Dorset House Publishing.

DeMarco, T. and T. Lister (1987). *Peopleware - Productive Projects and Teams*. Dorset House Publishing.

Dybå, T. (2002). Enabling Software Process Improvement: An Investigation of the Importance of Organizational Issues. *Empirical Software Engineering 7*(4), 387–390.

Edwards, S. H. (2001). A Framework for Practical, Automated Black-box Testing of Component-based Software. *Software Testing, Verification and Reliability 11*(2), 97–111.

Eickelmann, N. and J. Hayes (2004). Quality time - New Year's Resolutions for Software Quality. *IEEE Software 21*(1), 12–13.

El Emam, K., J-N. Drouin and W. Melo (1998). *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*. IEEE.

El Emam, K. and N. Madhavji (1999). *Elements of Software Process Assessment and Improvement*. Wiley-IEEE.

Erdogmus, H., J. Favare, and W. Strigel (2004). Return On Investment. *IEEE Software 21*(3), 18–22.

ETSI (2001). *ETSI ES 201 873-1: The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. V2.1.0, 2001-10*. ETSI.

Fenton, N. and S. L. Pfleeger (1997). *Software Metrics*. PWS Publishing Company.

Fewster, M. (1993). Test Automation Using an In-house Testing Tool: A Case History. In M. Kelly (Ed.), *Management and Measurement of Software Quality*, UNICOM SEMINARS, pp. 193–204. Unicom Applied Information Technology.

Fewster, M. and D. Graham (1999). *Software Test Automation*. ACM Press, Harlow, UK.

Fraser, S., D. Astels, K. Beck, B. Boehm, J. McGregor, J. Newkirk, and C. Poole (2003). Panel: Discipline and practices of TDD: (Test Driven Development). In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, languages, and Applications*. ACM.

Freedman, R. (1991). Testability of Software Components. *IEEE Transactions on Software Engineering 17*(6), 553 – 564.

Gao, J. Z., J. Tsao, and Y. Wu (2003). *Testing and Quality Assurance for Component-Based Software*. Artech House Computer Library.

Gilb, T. (1988). *Principles of Software Engineering Management*. Addison-Wesley.

Glass, R. (1994). The Software-Research Crisis. *IEEE Software 11*(6), 42–47.

Glass, R. (2004). Some Heresy Regarding Software Engineering. *IEEE Software 21*(4), 104–107.

Gopal, A., M. Krishnan, T. Mukhopadhyay, and D. Goldenson (2002). Measurement Programs in Software Development: Determinants of Success. *IEEE Transactions on Software Engineering 28*(9), 863–875.

Grady, R. (1992). *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall, Upper Saddle River, NJ.

Graham, D. (2001). Software Testing. In J. J. Marciniak (Ed.), *Encyclopedia of Software Engineering*, pp. 1330–1353. John Wiley & Sons.

Gray, E. and W. Smith (1998). On the Limitations of Software Process Assessment and the Recognition of a Required Re-orientation for Global Process Improvement. *Software Quality Journal 7*(1), 21–34.

Groth, R. (2004). Is the Software Industry's Productivity Declining? *IEEE Software 21*(6), 92–94.

Hambling, B. (1993). Realistic and Cost-effective Software Testing. In M. Kelly (Ed.), *Management and Measurement of Software Quality*, UNICOM SEMINARS, pp. 95–112. Unicom Applied Information Technology.

Harrold, M. J. (2000). Testing: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pp. 61–72. ACM Press.

Hayes, L. (1995). *The Automated Testing Handbook*. Software Testing Institute.

Henningsson, K. and C. Wohlin (2004). Assuring Fault Classification Agreement – An Empirical Evaluation. *Proceedings of the Symposium on Empirical Software Engineering*.

Hevner, A. R. (1997). Phase Containment Metrics for Software Quality Improvement. *Information and Software Technology 39*(13), 867–877.

Hicks, I., G. South, and O. Oshisanwo (1997). Automated Testing as an Aid to Systems Integration. *BT Technology Journal 15*(3), 26–36.

Hofstede, G. (1996). *Cultures and Organisations, Intercultural co-operation and its importance for survival, Software of the mind*. McGraw-Hill Education.

Howles, T. and S. Daniels (2003). Widespread Effects of Defects. *Quality Progress 36*(8), 58–62.

Humphrey, W. (2002). *Winning with Software*. Addison-Wesley, Boston MA.

IEEE (1988). *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE/ANSI Standard 982.2-1988*. IEEE.

IEEE (1990). *Standard Glossary of Software Engineering Terminology/IEEE Std. 610.12-1990*. IEEE.

Ince, D. (1993). Some of the Questions to Ask about Quality Assurance. In M. Kelly (Ed.), *Management and Measurement of Software Quality*, UNICOM SEMINARS, pp. 9–18. Unicom Applied Information Technology.

ISO (1991). *International Standard ISO/IEC 9126*. International Organization for Standardization (ISO), rue de Varembé, Case postale 56, CH-1211 Geneva 20, Switzerland: ISO.

Jakobsen, A. (1998). Bottom up Process Improvement Tricks. *IEEE Software 15*(3), 64–68.

Johnston, A. K. (1993). Muzzling the Alligators: a Pragmatic Approach to Quality. In M. Kelly (Ed.), *Management and Measurement of Software Quality*, UNICOM SEMINARS, pp. 19–30. Unicom Applied Information Technology.

Kaner, C. (1997). Pitfalls and Strategies in Automated Testing. *IEEE Computer 30*(4), 114–116.

Kaner, C., J. Bach, and B. Pettichord (2002). *Lessons Learned in Software Testing*.

Kaplan, R. S. and D. P. Norton (1996). *The Balanced Scorecard*. Harvard Business School Press, Boston, MA.

Kehlenbeck, A. (1997). Commentary: Automated test tools have not delivered on the quality promise...yet. *Software Magazine*, p. 3.

Krishna, S., S. Sundeep, and G. Walsham (2004). Managing cross-cultural issues in global software outsourcing. *Communications of the ACM 47*(4), 62–66.

Leffingwell, D. and D. Widrig (2000). *Managing Software Requirements - A Unified Approach*. Addison-Wesley, Upper Saddle River,NJ.

Leszak, M., D. Perry, and D. Stoll (2000). A Case Study in Root Cause Defect Analysis. In *Proceedings of the 22nd Int. Conference on Software Engineering*, pp. 428–437. ACM Press.

MacCormack, A., C. Kemerer, and M. Cusumano (2003). Trade-offs between Productivity and Quality in Selecting Software Development Practices. *IEEE Software 20*(5), 78–86.

Madsen, P. (2004). A Software Development Process based on Test-Driven Development and Testing by Contract. In *Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques*, pp. 225–240.

Martella, R. C., R. Nelson, N, and E. Marchand-Martella (1999). *Research Methods*. Allyn & Bacon.

Mast, J. (2004). A Methodological Comparison of Three Strategies for Quality Improvement. *International Journal of Quality & Reliability Management 21*(2), 198–213.

Mathiassen, L., J. Pries-Heje, and O. Ngwenyama (2002). *Improving Software Organizations: From Principles to Practice*. Addison-Wesley, Upper Saddle River, NJ.

Maximilien, E. M. and L. Williams (2003). Assessing Test-Driven Development at IBM. In *Proceedings of the 25th International Conference on Software Engineering*, pp. 564–569. IEEE Comput. Soc. Press.

Maxwell, K. and R. Kusters (2000). Software Project Control and Metrics. *Information and Software Technology 42*(14), 963–964.

McGregor, J. (2001). Testing a Software Product Line, Technical Report CMU/SEI-2001-TR-022. Technical report, Carnegie Mellon University, Software Engineering Institute.

McGregor, J. D. and D. A. Sykes (2001). *A Practical Guide to Testing Object Oriented Software*. Addison-Wesley, Upper Saddle River, NJ.

McQuaid, P. and C. Dekkers (2004). Steer Clear of Hazards on the Road to Software Measurement Success. *Software Quality Professional 6*(2).

Mosley, D. J. and B. A. Posey (2002). *Just Enough Software Test Automation*. Prentice Hall, Upper Saddle River, NJ.

Ngwenyama, O. and P. A. Nielsen (2003). Competing Values in Software Process Improvement: An Assumption Analysis of CMM from an Organizational Culture Perspective. *IEEE Transactions on Engineering Management 50*(1), 100–112.

Niazi, M., D. Wilson, and D. Zowghi (2005). A Maturity Model for the Implementation of Software Process Improvement: An Empirical Study. *Journal of Systems and Software 74*(2), 155–172.

Nicholas, J. M. (2001). *Project management for Business and Technology* (Second edition ed.). Prentice-Hall, Upper Saddle River, NJ.

Offen, R. and R. Jeffery (1997). Establishing Software Measurement Programs. *IEEE Software 14*(2), 45–53.

Patton, R. (2001). *Software Testing*. Sams Publishing, US.

Paulish, D. and D. Carleton (1994). Case-Studies of Software Process-Improvement Measurement. *IEEE Computer 27*(9), 50–57.

Paulk, M. C., C. V. Weber, B. Curtis, and M. B. Chrissis (1995). *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison Wesley Longman, Inc.

Pfleeger, S. L. (2001). *Software Engineering, Theory and Practice*. Prentice-Hall, Upper Saddle River, NJ.

Poston, R. (2005). Test Generators. In J. J. Marciniak (Ed.), *Encyclopedia of Software Engineering, DOI: 10.1002/0471028959.sof355*. John Wiley & Sons.

Pourkomeylian, P. (2002, March). *Software Practice Improvement*. PhD dissertation, Gothenburg Studies in Informatics.

Prikladnicki, R., J. Audy, and R. Evaristo (2003). Global Software Development in Practice Lessons Learned. *Software Process Improvement and Practice 8*, 267–281 (DOI: 10.1002/spip.188).

Rainer, A. and T. Hall (2002). Key Success Factors for Implementing Software Process Improvement: A Maturity-Based Analysis. *Journal of Systems and Software 62*(2), 71–84.

Rainer, A. and T. Hall (2003). A Quantitative and Qualitative Analysis of Factors Affecting Software Processes. *Journal of Systems and Software 66*(1), 7–21.

Rakitin, S. R. (2001). *Software Verification and Validation for Practitioners and Managers* (Second Edition ed.). Artech House.

Rasmusson, J. (2004). Introducing XP into Greenfield Projects: Lessons Learned. *IEEE Software 20*(3), 21–29.

Robson, C. (2002). *Real World Research* (Second Edition ed.). Blackwell publishing, Cornwall, UK.

Rubin, H. (1991). Measure for measure. *Computerworld, ABI/Inform Global 25*(15), pp. 77–79.

Shull, F., V. Basili, B. Boehm, W. Brown, P. Costa, M. Lindwall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz (2002). What We Have Learned About Fighting Defects. In *Proceedings of the Eight IEEE Symposium on Software Metrics*, pp. pp. 249–258. IEEE Comput. Soc.

SixSigma (2005). http://www.sixsigmainstitute.com/sixsigma/dmaic_6sigma.shtml, Last Accessed: 2005-02-04.

SW-CMM. (2005) SW-CMM, Software-Capability Maturity Model. http://www.sei.cmu.edu/cmm/cmm.html, Last Accessed: 2005-02-04.

Tanaka, T., K. Sakamoto, S. Kusumoto, K. Matsumoto, and T. Kikuno (1995). Improvement of Software Process by Process Description and Benefit Estimation. In *Proceedings of the 17th International Conference on Software Engineering*, pp. 123–132. ACM.

Teiniker, E. (2003). A Test-Driven Component Development Framework Based on the Corba Component Model. In *Proceedings of the 27th Annual International Computer Software and Applications Conference*, pp. 400–405. E. Teiniker and S. Mitterdorfer and L-M. Johnson and C. Kreiner and Z. Kovacs and R. Weiss.

Van Solingen, R. (2004). Measuring the ROI of Software Process Improvement. *IEEE Software 21*(3), 32–38.

Veenendaal, E. (2002). *The Testing Practitioner*. UTN Publishers, Den Bosch, NL.

Voas, J. (1997). Can Clean Pipes Produce Dirty Water? *IEEE Software 14*(4), 93–95.

Voas, J. (1999). Software Quality's Eight Greatest Myths. *IEEE Software 16*(5), 118–121.

Watkins, J. (2001). *Testing IT*. Cambridge University Press, Cambridge, UK.

White, L. (2001). Regression Testing. In J. J. Marciniak (Ed.), *Encyclopedia of Software Engineering*, pp. 1039–1043. John Wiley & Sons.

Whittaker, J. and J. Voas (2002). 50 Years of Software: Key Principles for Quality. *IT Professional 4*(6), 28–35.

Wohlin, C., M. Höst, and K. Henningsson (2003). *Empirical Research Methods in Software Engineering, In Empirical Methods and Studies in Software Engineering: Experiences from ESERNET, Editors: Reidar Conradi and Alf Inge Wang, Lecture Notes in Computer Science, LNCS 2765*, pp. 7–23. Spinger-Verlag.

Wohlin, C., P. Runesson, M. Höst, M. Olsson, B. Regnell, and A. Wesslen (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Springer-Verlag, Boston, MA.

Wohlwend, H. and S. Rosenbaum (1993). Software Improvements in an International Company. In *Proceedings of the 15th International Conference on Software Engineering*, pp. 212–220. IEEE Comput. Soc.

Appendix A

AFC Calculation method

This appendix provides an example of how to calculate AFC (Avoidable Fault Cost), i.e. through fictitious project data as applied in Table A.1. In the table, PF equals phase found and PB equals phase belonging, i.e. when the fault should have been found.

Table A.1: Example calculation of Avoidable Fault Cost (AFC), measured in hours.

PF / PB	Impl.	Function Test	System Test	Operation	**Total PB/phase**
Impl.	4*2-4*2 = 0	25*10-25*2 = 200h	18*25-18*2 = 414h	2*50-2*2 = 96h	**710h**
Function Test		15*10-15*10 = 0	5*25-5*10 = 75h	4*50-4*10 = 160h	**235h**
System Test			13*25-13*25 = 0	2*50-2*25 = 50h	**50h**
Operation				0	**0h**
Total potential/PF	**1h**	**290h**	**609h**	**355h**	**1255h**

Each cell in the table is calculated according to the formula in Equation 5.1 in Chapter 5. That is, the avoidable fault cost is calculated by multiplying the number of faults-slip-through in a phase with the excessive cost for a faults-slip-through between these phases. For example, if a fault on average cost one hour to find in phase ‘A’ but slipped through to phase ‘B’ where the average fault cost is five hours, the excessive cost for the fault is 5-1=4 hours. For example, in Table A.1, the dashed cell represents the AFC for faults that slipped through from implementation to Function Test. In the

dashed cell, the calculation states that 25 faults slipped through from implementation to Function Test and that a fault on average cost two hours to find in implementation and ten hours in Function Test, i.e. AFC = 25*10-25*2=200 hours.

Further, in the table, the most interesting cells are those in the rightmost column that summarize the total cost of faults in relation to fault belonging and the bottom row that summarizes the total unnecessary cost of faults in relation to phase found. For example, the largest improvement potential was in the implementation phase, i.e. the phase triggered 710 hours of unnecessary costs in later phases due to a large faults-slip-through from it. Further, System Test was in the example the phase that suffered from the largest excessive costs due to faults slip through to it (609 hours).

Appendix B

Result calculations

This appendix presents the calculations of AFC, improvement ratio and ROI in the case study in Chapter 5. Note that for confidentiality reasons, the values are only presented relatively, i.e. all the values have been divided by an anonymous factor X.

Comparison between projects - Product A

AFC (Release X)

Phase found / Phase bel.	FT	ST	FiT+6	Total potential/ belonging phase
Impl.	126	19	128	273
BNT	7,5	0	1,7	9,2
FT	-	2,5	35	37
ST	-	-	2,8	2,8
Total potential/ test phase	133	22	167	322

AFC (Release X+1)

Phase found / Phase bel.	FT	ST	FiT+6	Total potential/ belonging phase
Impl.	16	11,8	9,4	37
BNT	4,1	5,2	1,7	11
FT	-	2,5	1,0	3,5
ST	-	-	4,0	4,0
Total potential/ test phase	20	20	16	55

Project size (ReleaseX):	2916	(reported time)
Project size (ReleaseX+1):	1126	(reported time)
Improvement ratio (total):	2,3	((322/2916)/(55/1126))*
Improvement ratio (Impl.):	2,9	((260/2916)/(37/1126))*
Benefit (total):	69,0	(2,3*55-55)*
Benefit (Impl.):	68,6	(2,7*37-37)*
Investment cost:	11,9	(from questionnaire)
ROI (total):	4,8	((69,1-11,9)/11,9)*
ROI (Impl.):	4,8	((63,5-11,9)/11,9)*

Average fault cost/development phase

Phase found	Average cost/fault
Impl.	1,2
BNT	3
FT	9,8
ST	16
FiT+6	20

Abbreviations:

Impl. = Implementation+Unit test
BNT = Basic Node Test
FT = Function Test
ST = System Test
Fit+6 = Field Test+ 6 months

*: According to the formulas in equations 2-4.

Figure B.1: AFC and ROI for product A

Comparison between projects - Product B

AFC (Release X)

Phase found / Phase bel.	FT	ST	FiT+6	Total potential/ belonging phase
Impl.	47	25	19	91
BNT	2,7	3,9	3,4	10
FT	-	2,5	2,0	4,5
ST	-	-	2,4	2,4
Total potential/ test phase	50	32	27	108,1

AFC (Release X+1)

Phase found / Phase bel.	FT	ST	FiT+6	Total potential/ belonging phase
Impl.	50	18	19	86
BNT	7,5	7,8	1,7	17
FT	-	2,5	0	2,5
ST	-	-	1,6	1,6
Total potential/ test phase	57	28	22	107,5

Project size (ReleaseX):	791	(reported time)
Project size (ReleaseX+1):	1756	(reported time)
Improvement ratio (total):	2,2	((108,2/791)/(107,6/1756))
Improvement ratio (Impl.):	2,3	((87/791)/(83/1756))
Benefit (total):	132	(2,2*107,6-107,6)
Benefit (Impl.):	116	(2,3*83-83)
Investment cost:	16	(from questionnaire)
ROI (total):	7,3	((132-16)/16)
ROI (Impl.):	6,2	((111-16)/16)

Comparison within Release X+1 of Product B

AFC (for features *not* using the new concept)

Phase found / Phase bel.	FT	ST	FiT+6	Total potential/ belonging phase
Impl.	35	10	15	61
BNT	6,8	7,8	1,7	16
FT	-	1,9	0	1,9
ST	-	-	1,2	1,2
Total potential/ test phase	42	20	18	80

AFC (for features using the new concept)

Phase found / Phase bel.	FT	ST	FiT+6	Total potential/ belonging phase
Impl.	15	7,4	3,8	26
BNT	0,7	0	0	0,7
FT	-	0,6	0	0,6
ST	-	-	0,4	0,4
Total potential/ test phase	15	8,0	4,2	28

Project size (without):	899	(reported time)
Project size (with):	726	(reported time)
Improvement ratio (total):	2,3	(80/899)/(28/726)
Improvement ratio (Impl.):	1,9	(57/899)/(26/726)
Benefit (total):	37,1	(2,4*28-28)
Benefit (Impl.):	23,2	(1,8*26-26)
Investment cost:	16	(from questionnaire)
ROI (total):	1,3	((37,2-16)/16)
ROI (Impl.):	0,4	((20,4-16)/16)

Figure B.2: AFC and ROI for product B

Appendix C

Faults-slip-through Definition at Ericsson AB

C.1 Basic Test

- Basic stability problems that can be detected when testing components or sub-features in isolation (including component level load, performance, and endurance tests).
- Basic interface inconsistencies between components or within a sub-feature. That is, every component or sub-feature should work ‘stand alone’, including error cases.
- Faulty revisions in deliveries.
- Isolated faults in scripts and output text files including for example spelling faults and not understandable text.

C.2 Integration Test

- Basic software installation/uninstallation faults
- Faults in the current system installations that should work when starting each subsequent test phase
- Faults found in main function flows (smoke tests).

C.3 Function Test

- Every function should work ‘stand alone’ in a simulated environment (including human interfaces)
- The system should adhere to specified protocol standards.

C.4 System Test

- Load and stability faults (including performance, robustness, availability, load balancing, and reboot related faults)
- Faults found only when connected to “real” external systems.
- Complex multi-function faults.
- Restore and backup faults.
- Other installation faults, for example hardware related, data migration etc.