Mälardalen University Press Dissertations
No. 105

ON TEST DESIGN

Sigrid Eldh

2011

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

Copyright © Sigrid Eldh, 2011
ISBN 978-91-7485-037-6
ISSN 1651-4238
Printed by Mälardalen University, Västerås, Sweden

Mälardalen University Press Dissertations
No. 105

ON TEST DESIGN

Sigrid Eldh

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
fredagen den 21 oktober 2011, 13.00 i Delta, Högskoleplan 1, Västerås.

Fakultetsopponent: dr Eleine Weyuker, AT&T Labs Research

Akademin för innovation, design och teknik

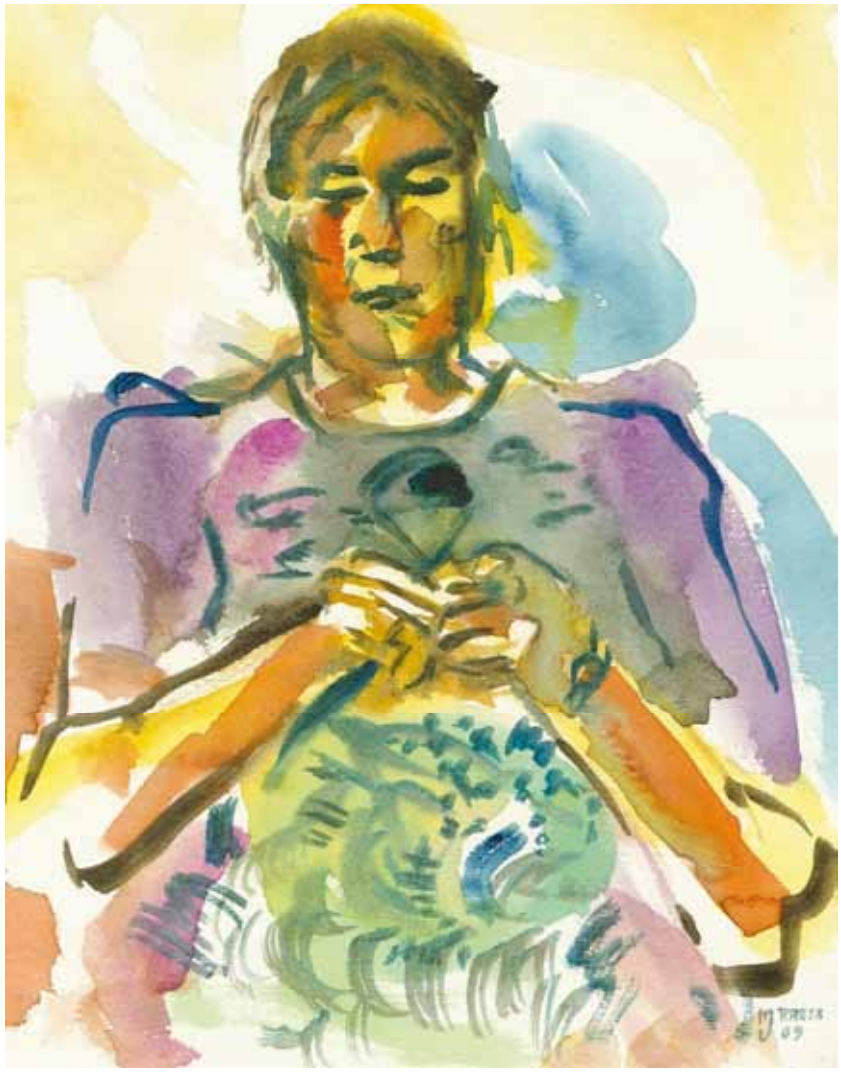
Abstract

Testing is the dominating method for quality assurance of industrial software. Despite its importance and the vast amount of resources invested, there are surprisingly limited efforts spent on testing research, and the few industrially applicable results that emerge are rarely adopted by industry. At the same time, the software industry is in dire need of better support for testing its software within the limited time available.

Our aim is to provide a better understanding of how test cases are created and applied, and what factors really impact the quality of the actual test. The plethora of test design techniques (TDTs) available makes decisions on how to test a difficult choice. Which techniques should be chosen and where in the software should they be applied? Are there any particular benefits of using a specific TDT? Which techniques are effective? Which can you automate? What is the most beneficial way to do a systematic test of a system?

This thesis attempts to answer some of these questions by providing a set of guidelines for test design, including concrete suggestions for how to improve testing of industrial software systems, thereby contributing to an improved overall system quality. The guidelines are based on ten studies on the understanding and use of TDTs. The studies have been performed in a variety of system domains and consider several different aspects of software test. For example, we have investigated some of the common mistakes in creating test cases that can lead to poor and costly testing. We have also compared the effectiveness of different TDTs for different types of systems. One of the key factors for these comparisons is a profound understanding of faults and their propagation in different systems. Furthermore, we introduce a taxonomy for TDTs based on their *effectiveness* (fault finding ability), *efficiency* (fault finding rate), and *applicability*. Our goal is to provide an improved basis for making well-founded decisions regarding software testing, together with a better understanding of the complex process of test design and test case writing. Our guidelines are expected to lead to improvements in testing of complex industrial software, as well as to higher product quality and shorter time to market.

ISBN 978-91-7485-037-6
ISSN 1651-4238



Konstnär: Johan Mauritzson

Finally I know that
I have much to say!
What I have discovered
Is now here to Stay!

from Fredrick S. "Fritz" Perls, In and out the Garbage Pail, 1969

Till min älskade Per

Om Test Design

Mer än hälften av svensk export utgörs av produkter som styrs av datorsystem. En stor del av kostnaderna för utveckling av dessa system spenderas på att säkerställa att programvaran fungerar tillfredställande. För många produkter rör det sig om miljardbelopp och motsvarar typiskt 30-70% av den totala utvecklings- och underhållskostnaden.

Testning är den dominerande tekniken för kvalitetssäkring av industriell programvara. Testning innebär att systemet testas i en kontrollerad omgivning för att upptäcka fel och avvikelser från det förväntade beteendet. Kärnan i testning ligger i hur man konstruerar testfallen. Trots att industrin lägger stora resurser på testning och har stort behov av bättre testmetoder är området förvånansvärt outvecklat.

I denna doktorsavhandling försöker vi öka förståelsen för hur man konstruerar och använder testfall och vilka faktorer som påverkar testresultatet. Den stora mängden av testtekniker gör det svårt för den enskilda testaren att skapa bra testfall. Vilka tekniker hittar flest fel? I vilken ordning skall man använda dem? Vilka för- och nackdelar har de olika teknikerna?

Avhandlingen bygger på tio studier och presenterar bland annat en metod för att utvärdera olika testtekniker. Metoden har lett till ökad förståelse av hur felorsaker bör klassificeras för att generellt kunna jämföra olika tekniker mellan olika system. Vi introducerar även en klassificering av testtekniker som utgår från teknikernas *effektivitet*, dvs. förmågan att hitta fel och täcka in systemet, *snabbhet* att förstå och skapa testfall, samt *användbarhet,* dvs. för vilka system och i vilka situationer tekniken är användbar. Vårt mål är att underlätta testningen och därmed erbjuda bättre stöd och en bättre förståelse för den komplicerade process som testfallskonstruktion och testdesign är. Våra slutsatser sammanfattas i konkreta riktlinjer för testning av industriella system. Genom att följa dessa riktlinjer ges möjligheter till avsevärda förbättringar vid testning av komplicerade system, vilket förbättrar förutsättningarna för ökad produktkvalitet och förkortad utvecklingstid.

Acknowledgements

At the conclusion of this PhD I will have climbed my own Everest, a journey that made me stop and enjoy the view, over and over. I often thought I had reached the top, only to understand I reached a local maximum. With me, on this fantastic intellectual and developing journey, was my very experienced tour guide, Hans Hansson, a superb supervisor and experienced research leader, who has not only been a strong motivator, but has always been available to patiently listen to my self-doubt and insights, and thoroughly reading any word I have written. Hans, you are so generous and insightful. Together on this journey has been my other outstanding supervisor and spiritual mentor, Sasikumar Punnekkat. Thank you for broadening my views. Along the way young fellow climbers like Daniel Sundmark appeared and gave important advice at crucial times. Without all of you – this mountain would have been insurmountable. Thank you from the bottom of my heart and the depth of my mind. Thank you goes also to Joachim Wegener, the opponent at my Licentiate Thesis defense, who asked just the right questions at the right time. In addition, I must sincerely thank my "opponent" to this thesis, Elaine Weyuker, for all her hard work and her lifetime of great papers in the field. Many thanks go to my examination committee: Tom Ostrand, Ina Schieferdecker and Robert Feldt. To my dear colleagues at the SAVE-IT program and to all personnel at IDT at Mälardalen University who have supported me, I give many thanks for all our good times together. These thanks also extend to my wonderful supportive colleagues at Ericsson AB; it is always exiting to go to work. In particular, I give my thanks to Lars-Olof Gustafsson and Michael Williams, who have served as my industrial mentors. For funding my research in collaboration with Ericsson AB and The Knowledge Foundation SAVE-IT program, I give my thanks. And thank you, my Master Thesis Students. In fact, teaching and being with you has been an encouragement. I must particularly mention the following students that have been instrumental to this thesis: Mats Larsson, Peter Jönsson, Hans Bokvist, Jörgen Stenmark, Adithya Gollapudi, Arvind Ojha, Guido di Campli and Savino Ordine. This thesis would not be complete without giving my creative and caring

family a special mention for all their support; my dear mother Gudrun, brother and artist Johan, aunt Viola and cousin Sebastian come to mind. Finally, my love and gratitude goes to you, my wonderful husband Per. My dearest, I am so blessed to share my life with you and every moment we have together.

Älvsjö, September 2011

Sigrid Eldh

Preface

The format and requirement of an academic thesis differs from that of e.g. a book or a course. Regardless, I have put extra effort in trying to view my main reader as a person from industry involved in creating software and eager to learn more about testing. It would be bold to claim that my findings are valid for all software systems, since there are always exceptions. Nevertheless, my goal has been that the research should target all, and not specifically telecommunication or middleware software. In fact, from a testing viewpoint, test design possesses similar problems, even if terminology and type of system varies. Test design is more important the more complex or quality demanding the software system is.

To guide the reader of this thesis the following may be helpful:

The first chapter should give a clear overview of the thesis for any reader.

A *Tester or Developer* should find particularly chapters 2, 13 and 14 most useful and straight forward, but there are a lot of tricks and hints on common problems relevant for developers and tester in most of the studies. The "Mistakes" study in Chapter 11 should be a "must read" for anyone writing a test case.

A *Manager* would probably enjoy the chapters mentioned above, where Chapter 2 is introductory, Chapter 13 is about test design techniques (and reasoning), and Chapter 14 is the plain guideline. In addition, we present a useful improvement method in our first study (Chapter 3) that could give some new insights.

For the *academic reader,* I have particularly attempted to keep the studies a similar look and feel, by providing initial summaries to each of the studies. Part III, the synthesis of my thesis, is more a proposal targeted to the industrial audience than a complete and fully validated academic effort. Rather, it may be viewed as a basis and inspiration for further studies. Despite its academic shortcomings, I did find it necessary to conclude and attempt a guideline for the industry that craves such information.

If you feel I have unjustly forgotten to quote any of your important work, I must beg your forgiveness in advance. Having read about testing for many decades, I do not possess a memory that traces all the

origins to my insights, some I cannot tell if they are my own or someone else's. Instead, be happy that you have the same conclusions or that you might have influenced me.

All faults herein are mine and my supervisors should not be blamed – in fact, they have been helpful in so many ways, far beyond the scope of this thesis.

With these final words, I hope you enjoy my thesis, as much as I have enjoyed making it.

Älvsjö, September 2011

Sigrid Eldh

Table of Contents

Part I
Introduction

Chapter 1. Overview of Research

1.1 Why Software Testing is Important

Software is ubiquitous, and if software malfunctions it can have devastating effects on society. Testing is the dominating method for quality assurance of industrial software, and is without doubt a costly standard practice in industry. Tolerance of faulty software differs in different industries and domains. We accept trains to stop, when a fault in the signaling system occurs. In telephony, we accept occasional loss of voice quality rather than to lose an entire call, but losing a single call is not considered to be a disaster. For example, if you are making a mobile call from a train and you lose your call, you can just call again in a few seconds. The impatience of a customer when they lose a call or when they are sitting still and waiting for a railway signal are both qualities indirectly related to testing and cost. We can build a system to be fault tolerant and "fail-safe" but it usually has its price. Furthermore, we now expect all types of devices – run by software – to be accessible instantly at our fingertips, wherever we are. To enable this sort of technology, the quality of the service is as important as the service functionality, thus – software testing not only ensures that a targeted quality can be met; it is a necessity to ensure that the quality is at a sufficient level.

Developing completely fault free software is almost impossible, but it is currently possible to produce very high quality software if you are prepared to accept the cost. Software testing is the preferred method used for assessing and ensuring the robustness of industrial software in large complex systems.

1.2 Brief Background

The test design technique (TDT) and the test design describe the very specific ways to construct test cases. The test design is in theory

general, but must always be applied or implemented for a specific goal of testing a specific system.

Determining how to perform test design efficiently and effectively is the main objective for this research. The software industry needs guidelines for test design to determine which TDTs are efficient, effective and applicable. In this thesis we set out to understand better what can be done at each level of test, and we identify the key factors that affect test design. We have explored manual and automated test cases, and seen which parts of the test process that could be re-defined to improve test execution. Obviously for a fair comparison of TDTs, all types of faults should be present in the experimental system in order to give all TDTs a fair chance to show their applicability.

Equally important is that by analyzing faults in real systems and understanding their frequency and distribution, we will be able to obtain more general results that can be transferred to other systems. As a side effect, we have also learned more about faults, fault-injection, and fault and failure distributions. The understanding of fault and failure distribution made us realize that these patterns are still an enigma, and a much-improved systematic fault categorization is necessary to improve TDTs. Currently the origin and propagation of faults is unique for a system – although the patterns of occurrence are still not apparent. Despite this missing piece of necessary information about types of faults that lead to observable failures, which we can find by test execution we still attempted to evaluate different TDTs based on the faults we did found. This leads to a less comparable result between systems, but we attempted overlapping TDTs in several studies, that this contributed to our conclusions. It is always useful to understand the properties of the system, not only faults, but other aspects like observability, type, programming languages and other software technologies used – which all in conjunction are specific factors on the software that make claims possible to generalize.

Our aim is to be able to compare TDTs, and to write efficient test cases using these techniques. By exploring the TDTs through a series of studies, we have – partly only based on our vast practical experience of industrial testing – been able to formulate guidelines for industry on test design that we are reasonable confident will improve the current practice of industrial test design. We also clarify areas where further research would be beneficial. Our goal is to make

claims that could be useful for all types of software systems, although there may be systems where our claims are not fully valid.
Related work is introduced in 2.5, for each study, and particularly for TDTs in Chapter 13, therefore we omit it from chapter 1.

1.3 Research Objective

The main research objective for this thesis work is:

To establish sufficient knowledge about test design to enable comparison of test design techniques and to obtain sufficient basis for defining guidelines for improved test design in the software industry.

Based on this objective our key research questions are:

1. Is the current state of knowledge about test design sufficient for making recommendations for testing in the software industry?

2. To what extent are different test design techniques utilized in industry – and if they are not, what hinders their use?

3. How could different test design techniques be compared?

4. If there are multiple ways to apply a test design technique, does the choice influence the coverage and efficiency of the resulting test cases?

5. What relations do system properties, e.g. detected faults and observability, have on test design?

1.4 Overview of Research Methodology

In this thesis we have used empirical research methodology. Below we summarize the different types of research methods in this area, based on [89] [136] [183][186][188][215][217].

Primary Research involves the collection and analysis of original data, utilizing methods such as

a) Experimentation (to test hypothesis), including treatments, outcome measures and experimental units, having the primary goal of testing the hypothesis or theory, and either being
   i) Randomized (or true) with initial random assignments
   ii) Quasi-experiments (lacking initial random assignments) where comparisons depend on non-equivalent groups. Note that [186] comments that "Quasi-experiments conducted in an industry setting may have many characteristics in common with case studies."
b) Surveys, "a retrospective study of a situation that investigates relationships and outcomes" [188] and is "the collection of standardized information from a specific population, or some sample from one, usually, but not necessarily by means of a questionnaire or interview" [183][186]
c) Case Studies (i.e. research strategy), is information gathering from few entities (people, groups organizations) that investigates a contemporary phenomenon within its real-life context, e.g. aims at deliberately covering contextual conditions and lacking experimental control. Can be either
   i) Single-case
   ii) Multiple –case
d) Action Research focuses on combining theory and practice [89] and "influence and change some aspect of whatever is the focus of the research" [183] as a distinction from the "case study that is purely observational" [186]. Can be either
   i) Iterative
   ii) Reflective
   iii) Linear

Secondary Research uses data from previously published studies for the purpose of *research synthesis.* Can be either

e) A Systematic Literature review (systematic mapping)
f) Meta-studies

Runeson and Höst [186] distinguish four types of purposes for research based on Robson's classification [183]:

I. Exploratory – finding out what is happening, seeking new insights and generating ideas and hypothesis for new research
II. Descriptive – portraying a situation or phenomenon
III. Explanatory – seeking an explanation of a situation or a problem, mostly, but not necessary in the form of a causal relationship
IV. Improving (i.e. emancipator) – trying to improve a certain aspect of the studied phenomenon

Finally the aspect of qualitative and quantitative data collection is often used in these types of empirical studies, where the qualitative involves words, descriptions, pictures, diagrams etc and quantitative data involves numbers and classes. "Quantitative data is analyzed using statistics, while qualitative data is analyzed using categorization and sorting" [186]. It is also common that mixed methods is used to provide better understanding of the studied phenomenon. To increase the precision of empirical research, triangulation may be applied [189][186]. Can be either

- Data (source) triangulation – using more than one data source or collecting the same data at different occasions
- Observer triangulation – using more than one observer in the study
- Methodological triangulation – combining different types of data collection methods, e.g. qualitative and quantitative methods
- Theory triangulation – using alternative theories or viewpoints.

The above method will be referenced in the overview of the empirical studies below. Further, details can be found in the beginning of the presentation of each study, including context of the study, research design and threats of validity. In Section 1.5 we further discuss the particular triangulation made in the different studies.

1.5 Research Scope

Another way to describe this research is to relate each study with each main research questions we attempt to answer. These research questions are not a complete picture of all research questions we have attempted to answer in this thesis, but summarize the main aspects. In Figure 1.1, we present the research questions addressed by the different studies.

Study	1	2	3	4	5	6	7	8	9	10
RQ1	x		x	x		x				
RQ2						x			x	
RQ3		x			x					
RQ4							x	x		
RQ5			x				x	x		x

Figure 1.1 Relationships between Research Questions and the studies.

In the following subsections, we elaborate on our main findings related our five research questions.

1.5.1 Research Question 1

Is the current state of knowledge about test design sufficient for making recommendations for testing in the software industry?

We have identified a gap in the knowledge between what is considered state of the art in test design and the comprehension of what can be measured and applied as general knowledge of software systems. We identified that these techniques where both overlapping in many different ways and often understood differently depending on the system under test. A plethora of TDTs exists, giving little help to industry. We have attempted to identify why the TDTs are not used in industry, and some other contributing factors for poor testing and ways to improve the area of test design. Establishing the level of know-how of techniques and the difficulties in translating this theoretical knowledge into actual test cases seems to be a major hurdle, thus focus should be on learning some major techniques, and having well defined test specifications and detailed enough test cases.

We have established enough knowledge to attempt a new taxonomy and make recommendations as a guideline for testing to the software industry that would be both efficient and effective, and furthermore applicable in most circumstances.

1.5.2 Research Question 2

To what extent are different test design techniques utilized in industry – and if they are not, what hinders their use?

We could establish that there is a very limited usage of but a few techniques in industry and that the positive techniques dominate. Major factors that hinder these are lack of knowledge (since we could establish that if you know a technique, there is little or no difference in the time to create and apply them), lack of system (software) understanding, and poor test case writing.

1.5.3 Research Question 3

How could different test design techniques be compared?

We could establish that we base most of our results on insufficient knowledge about how to make a fair comparison of TDTs: Small systems are often used, the technique is only compared with random or ad hoc results, or artificial faults are injected that are suitable for the technique, or not representing the real complexity where many faults contribute to observable failures. We have provided a rather thorough process in Study 2 (Chapter 4), with suggestions how to proceed in Study 3 (Chapter 5). We did apply parts of the process in Study 9 and 10 with a successful result.

1.5.4 Research Question 4

If there are multiple ways to apply a test design technique, does the choice influence the coverage and efficiency of the resulting test cases?

Since most TDTs define a rather large input domain or path, and not a single unique selection, the choice of how to apply them is very much

a human decision when it comes to the specific test case. It is important to understand that this the selection possibilities shrink with the techniques, and that is their main purpose. In fact, even techniques we do assume are defining uniquely specific data or path are often variable, depending on definition detail, the system, and the goal of testing. Therefore, it is very important to aid the tester as much as possible to clarify unique groups of input, which we assume/believe the system handles similarly, regardless of which individual we chose.

However, since real industrial software systems are inherently complex, it is possible that our assumptions are wrong. We can conclude that selecting test cases deliberately from different groups seems to challenge the execution and coverage more, and thus contribute more to better testing. Thus, this seems to be the best selection mechanism; techniques that have "too large" selection areas contribute less to selecting effective test cases, but might also be simpler to create, since there are more varieties allowed.

One can claim that TDTs that overlap many other techniques are easier to apply. Such techniques are often a name of a "family" of techniques, providing a wide range of selection possibilities, which make them more dependent on the human applicability for effectiveness, e.g. positive or negative TDTs. Furthermore, there is a large selection of either input data and/or execution path selection that fulfills these techniques making a human selection seem necessary.

1.5.5 Research Question 5

What relations do system properties, e.g. detected faults and observability, have on test design?

System properties impact the applicability of the techniques. For instance, if a system lacks branches in the code, branch covering techniques are inherently meaningless. If a fault does not propagate to an observable state in the execution (nor has any side effect) one can basically call that faults dormant, since it might affect a future version of the code. We only know there is a fault if we at some point have been able to execute the code, thus, established that it is a fault in the first place. Detected faults only mean that we have executed that particular path in a particular context, but this does not guarantee that

the code is free of faults in all contexts. This means that the number of faults is only related to the amount of testing we have performed, since calculating the complete possible ways and contexts to execute the code is currently impossible for real life systems. Thus, we cannot draw very much conclusions on the quality of our system based on this. This is why coverage, and in particular more advanced coverage provide a better view of quality for most systems, since it gives us a number to compare with. Still, all the faults can be in the last percent of 100% of that coverage – or outside the scope of what that particular coverage measures. It is possible, but very costly to improve on observability of a system, which is often built in to the system architecture from the start.

1.6 Overview of Thesis

We have divided the Thesis in three parts. Part I is the introduction, Part II presents the ten different empirical studies and Part III provides synthesis of these studies in the form of our systematic conclusion on the test design technique taxonomy and the Guidelines. Here we describe the overview of the thesis:

Part I - Introduction

Chapter 1 - Overview of research including research methodology, research questions, studies and contributions.

Chapter 2 - Introduction to Test Design, and introduce some necessary terminology, context and related work. Detailed terminology and related work can be found in each study, and for test design techniques, in Chapter 13.

Part II – Empirical Studies

Chapter 3 (Study 1) - Component test process improvement: The first study is a large industrial study based on an improvement project in industry, primarily intended to improve the quality of component test. The research method could be classified as action research of an iterative and partly reflective nature, since the researcher was

instrumental in both defining, implementing and concluding the results. The most sever research design problem, was the lack of systematic recording of the observation, though a large amount of data was collected, aggregated, analyzed and used within the project.

This study provides the motivation for this thesis, and includes a large list of "lessons learned". The way this study was done make it difficult to share the case study data, which was not in detail made available outside the company. Some of the conclusions on the impact were made outside the researcher's control, strengthening the final result. By this extra external review observer triangulation can be said to be deployed. In particular, the external review targeted the improved fault levels for the different design teams and thus confirming the researcher's quality improvement findings. Since the test process improvement was deployed at 22 different design teams, there is also a data source triangulation. The study consists of several aspects addressing processes, organization of test and assessment of quality and evaluation of results. It also contributes to the know-how of how to judge and utilize coverage, static analysis and test harnesses. The study was initiated 2003 and concluded 2005, but was not reported until 2008 at ISSRE [62].

Chapter 4 (Study 2) – A Framework for comparing Test Techniques This study developed a framework for comparing TDTs. The research method is a primary research, doing theoretical formation, based on synthesis derived from literature studies, where we aimed to describe a way to measure comparison of TDTs. The main contribution was a detailed process and related measurements describing some of the important aspects, in particular *efficiency, effectiveness and applicability.* These three aspects are highlighted in the studies that followed. This study was presented at IEEE TAIC-PART 2006[65].

Chapter 5 (Study 3) – "Component Testing is not enough" is our Failure-Fault case study. This is a descriptive single case-study focused on telecom middleware systems. We aimed to implement the first sub-process of the process described in Chapter 4, with the goal to identify faults that we could re-inject to our system. We propose a new classification of faults synthesizing existing classifications. This case study led to a deeper understanding of fault analysis and showed

why this step is almost always omitted when comparing TDTs. It also showed why this is the main hurdle preventing sufficient TDT juxtaposition. We learned a lot about complex fault-failure relations, why unit test is not sufficient and what the real characteristics of faults are. This gave us a much better understanding of the difficulties in understanding the *effectiveness* of a TDT. We did not manage the fault-injection objective. This study significantly changed our pursuit and made us rethink our approach. The study was presented at TESTCOM-FATES 2007[66]. This study has been replicated, where the results was confirmed by another researcher for another system [81].

Chapter 6 (Study 4) – This is an industrial single-case study on *applicability* of TDTs. We investigated characteristics of the techniques used and proposed new approaches attempting to systematically investigate the benefits of some TDTs in an industrial setting. We had no control over the data collection, which was done by the industry testers. While setting up the study, we found some applicability issues, and no faults were found, causing a debate if the system under test was just fault-free where we tested, or if an explanation was how the TDTs were applied. It turned out that it was how the technique was applied. The industry decided anyhow to make a second attempt with a different system, believing it would be more valuable to test a more recently developed system. We obtained important data from the second attempt. However, the new results were not possible to compare with the previous attempt, since both the method and the system under test had changed. Instead we had to treat this not as an experiment, but could only focus on the latter data result.

Chapter 7 (Study 5) – Three investigations on *applicability* of test design are performed among three different student groups. We tried out a number of ways to measure how students used TDTs, thus method triangulation was used, in addition to some aspects of data triangulation, particular using different sources and occasions. The research design and method improved for each of the experiments, and better control was enforced in the data collection. We repeated similar sets of questions for three years and with varied results. This

study shows the problems of understanding *applicability* and documentation of test cases. This study provided insight into design studies for teaching TDTs. The evaluation of measurements collected inspired the improvement process based on the mistakes study in Chapter 11, which describes the systematic mistakes made by students.

Chapter 8 (Study 6) – Industrial experiments on TDTs, conducted by a series of groups, each consisting of approximately 10 people on different occasions and in different industries, which provides data source triangulation. In these multiple experiments we collected data during a 6-month period, where small refinements were made between each trial. This study focused on TDTs and the actual *applicability* of them. The experiment was set up measuring pre- and post knowledge, with a treatment in between, where TDTs were taught in an abbreviated manner. The data was collected in a more quasi-experimental approach, including qualitative survey for the pre- and post study, with a treatment (a short test seminar) in the middle. In addition a practical element that demonstrates the technique could be applied to strengthen the data. Since one can question the researchers' interpretation of the TDT usage, we differentiated the result in three categories: Clearly not using the technique, clearly using the technique and hesitant usage of the technique. One important conclusion is that experienced developers and testers have problems with test design, and that the know-how of test design is very limited in industry. The data has been statistically treated. This study uses a method triangulation, in contrast to Chapter 7, which only measure applicability by delivering the test cases on demand.

Chapter 9 (Study 7) – This chapter consists of two parts. First a synthesis of negative usage techniques, so called attacks, is made. Then we present an industrial single case-study on *applicability* of negative TDTs in conjunction with a series of other TDTs. The framework described in Chapter 4 was partly used, and provided a series of conclusions and insights particularly on negative TDTs, but also confirming some of the techniques used in both the industrial studies in Chapter 8 and Chapter 10. There is data triangulation with

Chapter 10, using different systems, and different persons collecting the data.

Chapter 10 (Study 8) – This experiment aimed to measure *efficiency* and *applicability* of TDTs using parts of the framework on fifteen different open source systems. The case study's main goal was to collecting data on efficiency, timings and fault finding. We wanted to collect further insight on applicability on a wide variety of system types, while keeping the method as similar as possible. This provides a data triangulation. A series of techniques was explored and then followed by repeating the same approach on the same systems and measuring different coverage techniques on four different coverage measurement systems. Repeating the test cases with coverage provide a check on the repeatability of the experiment. Then new test cases were created based on the existing coverage information, with the goal to increase coverage. This approach allows testers to utilize coverage as a complementary technique and seems to improve the overall test result (and fault finding ability).

Chapter 11 (Study 9) – Systematic Mistakes made when writing test cases and measuring applicability of TDTs. This chapter can be viewed as a result of the third experiment presented in Chapter 7 and then complemented by interviews in industry, to form a new theory. This means that some attempt was made to make data triangulation. This study provides some insight in why test design is not applied efficiently, and also contributes by showing how to make an improvement process based on these mistakes. This study was presented at ICST 2011[70].

Chapter 12 (Study 10) – Automation of the test management process. This is a single case-study performed in industry, including

a. Setting up of test case connected to software items
b. Support for analysis of auto-locating faults
c. Automatic software build, test case execution and regression testing
d. Instant test progress reporting

The above are all features that contribute to removing the test manager from the execution phase. This study was a specific example from industry of direct *efficiency* improvement. This study was presented at ICST 2010 [68].

Part III – Synthesis: Taxonomy and Guideline for Industry

Chapter 13 – Taxonomy of Test Design Techniques, including families, "subsumes" hierarchies, synonyms, variants and groups of techniques for different goals. The main goal with this chapter is to ease the comprehension and future juxtaposing of TDTs – within families and across families. This chapter also includes related work to TDTs, in particular Vegas Schema [201][203], Ammann and Offutt's recent attempt [4] and Beizer's seminal impact[15] of the area.

Chapter 14 – Guidelines for Industry on Test Design Techniques. This is a proposal for how we suggest test design could be utilized and used efficiently and effectively. We also provide suggestions of TDTs that are easily applicable for (most) industries, but presented in an order, depending on the quality goals set, thus it is possible to "stop" at any level of improvement. These suggestions assume full comprehension of the technique. We also addressed other contributing factors that impacts improvement of the test design, e.g. better specifications. Finally, we provided a simple "self-test" to use as an aid for improvements on test design.

Chapter 15 – The conclusion and summary of the thesis includes a section on use for industry and for academia respectively – and thus defines a share of the future work.

The ten studies in Part II are not described in a strict temporal order, since some of the studies were conducted in parallel and over longer duration (several years). We have instead focused on an order that would aid the presentation in this thesis.

We want the reader to note that the related work to the goal of this thesis is rather lengthy and in Chapter 13, but are also available in direct relation to each study.

1.7 Contributions

This thesis contains the following main contributions:

1. It provides an account for the actual status of test design in industry, explains some problems in utilizing and realizing test design in industry, and suggests improvements that may facilitate more efficient and applicable approaches. Compared to the state-of–the art, probably best captured by Bertolino [24], our approach provides novel insights that challenge some of her fundamental claims. Not only do we remove the limiting view of black-box and white-box, but also discard the diminishing factor of levels of test. Instead we provide comprehensibility (knowledge) as a discrepancy factor of applicability of techniques.
2. It provides a taxonomy and framework for comparing efficiency, effectiveness and applicability of different TDTs, thereby improving on Vegas Characterization Scheme [203], as well as among a plethora of others in the field. Compared to earlier works our Taxonomy aims to simplify learning and improve on industry use of test design techniques. A detailed related work is described in sections 2.6 and 13.7. Our contribution on comparisons challenges Bertolino's [24] suggestion to investigate different domains and instead propose to investigate faults and their propagation patterns as a viable distinguishing factor.
3. It provides a novel set of guidelines for improvement of industrial testing practices, which extends current standards and recommendations (e.g. [110][128][129][4][203]) by concrete suggestions on how to use the above framework in the selection of appropriate TDTs.

Overall, the thesis identifies gaps between academia and industry by applying scientific measurements, theory, and methods in a set of empirical studies, as well as provides concrete measures for improvement through the suggested taxonomy and guidelines. The more detailed contributions on a per chapter level are as follows:

Chapter 3 – Component Test Process Improvement

A method for driving quality improvement with five quality levels, including applying the method at Ericsson and evaluating obtained

quality improvements. The main finding is that even 100% statement code coverage does not necessarily result in improved quality of the test cases. However, statement code coverage can successfully drive improvements of complex software if combined with additional testing, static analysis and code review.

Chapter 4 – A Framework for Comparing TDTs

A detailed process is described on how to compare TDTs, including definition of important aspects and measurements, in particular the concept of *applicability.* The main findings are that a fair comparison of effectiveness of TDTs assumes that a series of different types of faults are injected and propagates to the interface (are observable) and efficiency includes all aspects of the test case construction, not only test execution.

Chapter 5 – Component Test is not enough

A new taxonomy for faults is proposed and used to classify an industrial middleware system, based on the first sub-process in the previous study. The main findings are that

- There is a complex relation between code faults and the failure that a test case can expose in execution, e.g. more than 34% of the failures for the specific system had more than one fault location (file) origin.
- It is difficult to identify true re-injectable code faults that can be used to compare TDTs, hence measure *effectiveness* of a test design as a generic statement.
- The type of faults found will be related to the selected test execution used for the system under test.

Chapter 6 – Industrial experiments on *applicability* of Test Design Techniques

Industrial studies pose a series of problems if detailed scrutiny is not possible. The main findings are that

- The Boundary Value Analysis TDT is not easy to comprehend and apply

- Using TDTs improves the fault finding in both system under test and the automated test tool

Chapter 7 – Know-how of Test Design Techniques

These experiments establish the difficulties to measure comprehension and applicability of TDTs and to document test cases sufficiently. The main findings are that

- Positive techniques including state-transition techniques, seems easier to comprehend than equivalence partitioning, boundary values and negative TDTs.
- TDTs are more related to comprehension levels than software integration levels

Chapter 8 – Industrial surveys on test design techniques

The experiment concludes the limited knowledge and usage of TDTs in industry. The main findings are that

- The know-how of TDTs among experienced practitioners (testers and developers) is below 10%.
- Positive test seems to dominate the test case execution
- Boundary Value Analysis is a known name of a technique, but few practitioners can apply it correctly

Chapter 9 – Industrial use of Negative TDTs

An initial attempt to classify and generalize negative TDTs is proposed. The main findings are that

- Only 7 of 23 negative “Attacks” could easily be translated into test cases applicable for the selected specific industrial system.
- Negative testing is an insufficiently studied family of techniques

Chapter 10 – Measurements of the *efficiency* and *applicability* of TDTs on fifteen open source systems

Utilizing our framework, we could establish that overlapping TDTs takes equally long to apply, and the difficulty seems to be finding "the next" area of application. The main findings are that

- TDTs involving "outside the boundary" (negative approaches) seems to find more failures
- Code Coverage is an effective technique as a complement to functional TDTs

Chapter 11 – Systematic Mistakes of Test Cases Writing

We propose a new process for how to improve test cases, based on identifying common mistakes. The main findings are that

- TDTs are not applied efficiently due to lack of detail, comprehension and mistakes done at the test case writing (the "implementation") of the TDT.
- Poor testing and costly test automation seems to be caused by poor test case writing.

Chapter 12 – Automation of the Test Management Process

We provide the contents of Test Management Systems and expand the automation scope, thereby enabling savings in managing the test execution phase, particularly debugging (fault location), regression testing and test management. The main findings are that

- Setting up test cases connected to software items enables support for analysis of auto-locating faults.
- Correct automatic support can save money and time, and are making test managers redundant in the test execution phase

Chapter 13 – Taxonomy for TDTs

A new taxonomy for TDTs is proposed that defines groups and classifies individual TDTs for the ease of comprehension

- Synonyms, variants and subsumes hierarchies of TDTs are suggested

- New structures of TDT's are proposed

Chapter 14 – Guidelines for Industry on Test Design

This chapter proposes an order of how to design your test cases for best effectiveness, efficiency, and applicability. This proposal concludes that

- It is possible to suggest an order of TDTs to be used for improvement of testing
- It is possible to provide a series of improvement suggestions that would facilitate improved test case creations and utilization of TDTs.

Contributions to this Thesis

Sigrid Eldh is the main author of and main contributor to all papers related to this Thesis (listed at page 385). Hans Hansson and Sasikumar Punnekkat contributed in their supervisory roles. Contributions in the different studies have been in form of discussions and reviews of the manuscripts: Daniel Sundmark, Anders Pettersson and Joachim Brandt) or tool construction (Mark Street) and in addition from the Master Theses work primarily data collection (Peter Jönsson, Adithya Gollapudi, Arvind Ojha, Savino Ordine, Guido Di Campli). Peter Jönsson was also instrumental in the analysis and construction of the Fault Taxonomy used in Study 3, Chapter 5.

Chapter 2. Introduction to Test Design

In this thesis we focus on *test design*, specifically test design techniques (TDTs), which is the core of testing. Applying a TDT to a system, results in a test case used to execute the system and get a verdict. Testing has multiple goals, e.g. ensuring that specific paths execute correctly, but also attempting to find faults. Furthermore, testing is itself a way to measure some qualities of the system, by combining several test case executions in suites that exercises the system. This can result in additional information about the system, in terms of measures of the code coverage, performance, robustness, usability, functionality or other aspects and qualities of a software system. The main difference between just executing the system and testing is that the test case must include a verdict. A *verdict* means a way to determine if the system behaves as expected with respect to the specific aspect and context or if the system fails. *Failing* means that the expected service is not delivered, or the system does not behave according to expectation, specification or some defined measurement or norm.

2.1 Expectations on Testing

Our goal is to show that increased know-how of testing makes it possible to produce high quality software, without increasing the cost. Unfortunately the trend in industry is the reverse, focusing more on producing software fast than on quality. There is, however, a limit on how bad quality any user can accept.

Many customers expect fault-free software, which is unrealistic in large complex systems. In most software systems, it is possible to minimize unscheduled and unwanted runtime stoppages, ensuring continued execution. The impact of faults propagating into failure of a service can be limited through other techniques, such as self-correcting code, layered protocols, using redundancy or other fault-tolerant computing techniques, but this is not discussed in this thesis. In fact we do not know enough about which faults propagate into a

visible failure, or which is the best way to handle software failures. Many such adjacent areas have been encountered in our research and we report them only to the extent we have investigated them.

We define our terminology below and provide information on some aspects of the execution of the test cases. Although this is not the main area of this thesis, the result of the implemented test design has great impact on the test execution.

In particular, three aspects of effective, efficient and applicable testing have been in focus. These aspects explain how test and test design is dependent on the software development process:

- **Efficiency:** The speed with which we can create test cases for a specific system, and the speed with which can we apply, execute and evaluate (determine the result of) these test cases. Efficiency is not restricted only to the test execution; we take the more practical standpoint that efficiency concerns all activities of the test process.
- **Effectiveness:** The ability to provide new or added coverage**.** Effectiveness can be viewed as the fault finding ability (in relation to system and specification) of the test cases.
- **Applicability:** When (in what system) can a specific test design (a specific test case) be applied and under what circumstances is it meaningful**.** Applicability also refers to the ability to transform a theoretic (and thus general) approach to a specific situation (make an implementation).

Test Design is mainly influenced by the requirements, the system design, the actual code, and the execution paths. The impact on the test design concerns what level of test is addressed and the scope of the system addressed. In turn, the result of the test design process impacts the test cases. Other aspects of the test design process are only implicitly discussed. This includes, methods to define and handle test cases and test case execution, the test implementation resulting in test code, the test tools (if any) and the evaluation of the outcome of the test execution, which results in the measurement of the system quality.

Figure 2.1 and Figure 2.2 illustrate how we limit the scope we are addressing. In Figure 2.1 we identify the targets of this research, indicated by green. Semi-dashed (dot dash or light blue) boxes are

partly or implicitly addressed (e.g., (Test) execution), and dashed boxes (pink areas) indicate areas that are not considered at all (e.g., Test Code).

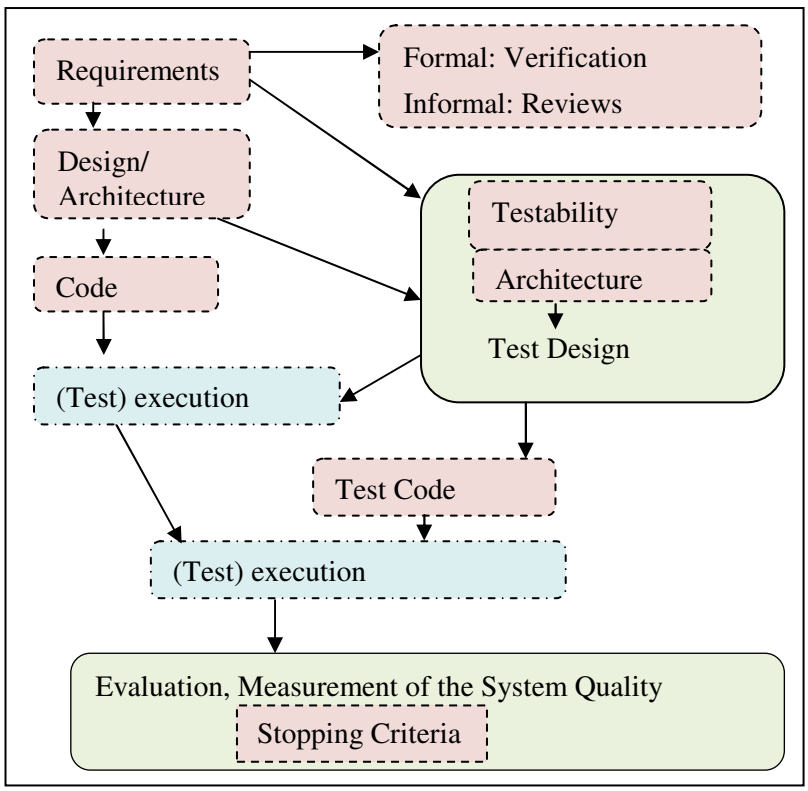


Figure 2.1 Overview of targeted areas in this thesis.

In Figure 2.2 we describe test design from a different perspective, through the efficient, effective and applicable view. This is divided into TDTs and the test design implementation. TDTs can be divided in many ways. We will use a categorization of techniques into:

1. Functional
    1.1. Input/parameter related
    1.2. Path/graph or order related (Structural)
2. Non-functional

Functional testing aims to test some aspect, function or feature of the system. Functional techniques can be divided into input and path specific use of the technique – both are needed when developing the

test case, and thus the specific execution flow. Even if you need to specify both input and path (or flow) in the test case, the selection of which aspects is the most important (or first in order) defines the goal of the test and thus defines the technique.

This means if we focus on a certain structure (for example a specific path in the code) we must define the input accordingly to achieve this goal. If the goal is to make sure we have a test case for all types of input, our goal looks different and so does the test case. It is common to describe Functional (as solely being about input selection) and Structural techniques as separate categories. This can be seen in e.g. ISTQB [161].

Non-functional aspects (characteristics), are based on a measurement that is analyzed in some form, usually using a series of combined functional executions of the system (e.g. performance, load), or other aspects or abilities (e.g. usability, installability etc.) of the system. We will discuss this further in Chapter 13.2.5.

Note that the historical “black-box” and “white-box” view of software can be applied to TDTs, but the unfortunate confusion with these techniques and “level” of testing is often more prevalent. Some techniques are black-box (input related) and some are white-box (structural, but only on code level). Most techniques are applicable at any level of testing so the black-box and white-box view when it comes to testing have lost its significance. Not only does the understanding of TDTs and how to apply them have significant meaning for the result, but also it is important to make sure that no unnecessary limitations on where to apply these techniques are introduced.

Another distinction is the implementation of the test design, where we distinguish between: “manually” – (a written text description of a test case for human use), versus the “automatic” or “rule-based” (which could be interpreted as computer generated test cases which could be executable code or generated code). These two aspects affect both the test design process in itself (how test cases are constructed), and the resulting implementation, the executable test case including evaluation. With these two main distinctions in mind, there are totally different aspects of test design. In this thesis we have particularly looked at functional test cases focussing on many techniques, which address both execution path and input/output, with a tendency to look

at primary input techniques. We have looked at the differences between the stages of the test case development and evaluation and tried to categorize them according to rule-based techniques or "human" techniques. In this thesis the main focus is TDTs for functional tests.

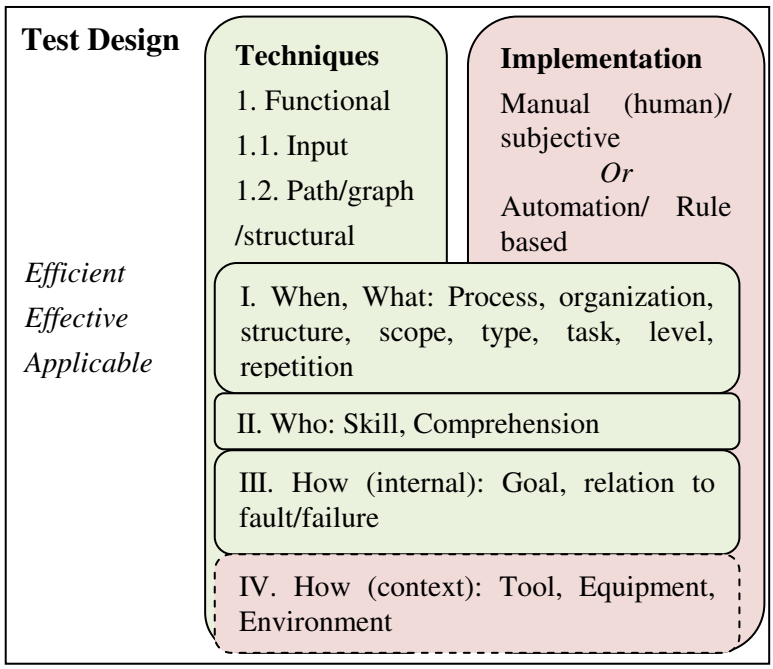


Figure 2.2 Targeted areas of Test Design: Another perspective

In industry, it is common to talk about test design implicitly and instead address the area through one of the following key words:

I. When, What: Process, organisation/structure, scope, type, task, level, repetition (frequency)
II. Who: Skill, Comprehension
III. How (internal): Goal, relation to fault/ failure
IV. How (external): Tools, equipments, environment

One can claim that all these aspects I-IV test process influences on how test design is done and what test cases are implemented and selected. For example in what phase of a process the test case is created (I) and with what goal (III) might be totally separated from

how often the test execution is done, with what tool (IV) and in what context (IV).

2.1.1 Test Design Techniques and Test Cases

Test design is a phase with the goal to create test cases. A *test case* is the result of applying a test design technique to a specific location in the software system application (or part thereof). Some test design techniques will, when applied, result in a set of test cases for a specific part of a system. By a *test design technique (TDT)*, or test technique for short, we mean a method or approach that systematically describes how a set of test cases should be created (with what intention and goals, and possible based on rules). The TDT aids in limiting the number of test cases that can be created, since it will be targeting a specific type of input, path, fault, goal, measurement etc.

Furthermore, a *test case* is defined here as a repeatable execution in the system, with a specific start (i.e. location in the system), a step by step description of the particular execution (with appropriate and exact input) and an expected result. We count each unique new input as a new test case. A test case can also be written in a generic manner, where input and its corresponding evaluation or "output" used to determine the result of the test (the verdict), is separated from the execution flow. We call these *generic* test cases. Test cases should be described in a way that allows them to be automatically executed. A test case can be as simple as pressing a button but can also comprise several pages of instructions. We do not make a distinction between a test case and a test procedure as in IEEE Std. 829 [110]. Thus a test case should be explicitly executable regardless of its representation (human readable text or programming language). A test case must have a unique identifier and a reference to documentation or requirement to ensure traceability. A *verdict* is the result of applying and executing the test case in the system. The verdict information is stored in a *test record* at the time of execution. A *test case* should be *repeatable* by anyone (yielding the same result) and measurable, by recording deviations from expected result. A *test suite* is a series of consecutive *test cases* that may or may not have anything in common. In a test suite, test cases may be dependent on each other or

independent on previously executed test cases. The IEEE 829 standard [110] calls the relation between test cases as *inter-dependencies*. All test cases need a "starting point" before they are ready to execute. This can either be the same for all test cases or, more commonly, a series of test cases defines different paths through the system to get to a "starting-point" for another test case.

2.1.2 Efficiency

Efficiency is defined by Rothermel and Harrold [185] as the measurement of the computational cost, and determines the practicality of a technique. We believe, however, that efficiency must be considered in a broader context. We expand their definition to include both execution and the *creation* of the test case. This includes time required to understand and implement the test case using a specific TDT. *Efficiency* of a *test design technique* is how fast the technique is understood, how fast the location where to apply the test case is identified and how fast the test case can be developed. The efficiency of a *test case* is how fast you can execute it and determine a verdict. *Efficiency* of the *test case* is closely related to automation. We often focus on test execution, which is the most obvious saving. Many aspects of test case creation can be automated. All execution of test cases can be automated, but the cost of automation of some of the test cases is not always justified.

2.1.3 Effectiveness

Effectiveness of a technique can be defined as the number of faults the technique can find. This is a concept explored by Rapp & Weyuker [181], who states that "effectiveness of a test technique is only possible to measure if you can compare two techniques for the same set (i.e. software), but the result is not general", meaning, that it is only valid for a specific set. We aim to find ways to make two techniques comparable and the result general and define some theoretical limitations on juxtaposing the techniques. We define an *effective test case* as a test case with the ability to find/expose faults (failures) or a test case that improves the coverage of the software execution paths. All initial test cases are therefore effective in the

beginning, since they initially add some coverage. Judgment of what test cases are effective should be done given a set of test cases, by comparing if the test cases have the same coverage or additional coverage.

2.1.4 Applicability

Applicability of a technique, relates to the efficiency of the TDTs, and adds a dimension of meaningfulness. It should be possible to develop a test case based on a specific technique, within reasonable time and cost. Applicability becomes a combination of the difficulty to learn, use, create and evaluate the result of test cases with a specific technique for a particular system. Applicability also encompasses the ability of the technique to be automated, describing and minimizing the human intervention, and corresponding to a well defined "rule", which makes the TDT an unambiguously repeatable process for each new test case applied to the software system.

One aspect of applicability is *generality* which measures the ability of a technique to handle realistic and diverse language constructs, arbitrarily complex code modifications and real applications. If a technique is not general, it is only valid in its specific context and not necessarily possible to apply to other software and domains [185].

2.1.5 Fault, Error, Failure

The related terminology in this area (fault, error, cause or reason, failure, bug, defect, incident, and anomaly) is often confusing because these terms are used interchangeably and inconsistently by many both in industry and academia; see further discussion in Mohaghegi et al [156]. Therefore we define the following terms inspired by earlier work of Avižienis & Laprie [8] and Thane [196], where a *fault* is the static origin in the code, that during dynamic execution propagates, (in Figure 2.3 described as by a solid arrow) to an *error* (which is an intermediate infection of the code). If the error propagates into output and becomes visible during execution, it has caused a failure. An error is thus the manifestation of a fault *in the system* and a failure is the effect of an error *on the service.* An error or failure can both cause another error to occur**,** or trigger another fault. At Ericsson, failures

are reported as Trouble Reports (TRs). Occasionally these TRs, in their analysis section, give a more direct explanation of the cause of the failure, but mostly they just describe the symptoms. *TRs do not uniquely identify failures (i.e., several TRs may identify the same* failure). There is not a one-to-one relationship between a fault and a failure (i.e. different faults may lead to the same failure and some faults may cause multiple failures).

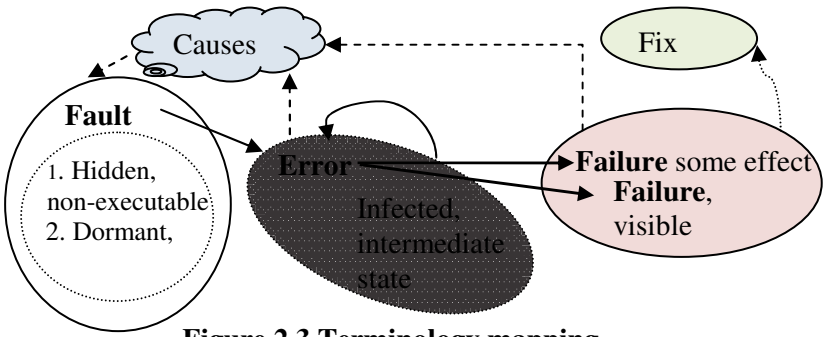


Figure 2.3 Terminology mapping

A failure can, in turn, propagate to another part of the software and be the cause of another error or failure**.** One difference compared with Avižienis & Laprie [8] is that we separate the actual cause of the fault from what manifests itself in the software. For example, an incorrect specification can lead to a fault in the software. To find what code changes are needed (see fault-injection technique in 10.3.1) in order to represent such a fault is not clear. It may be that the fault specification defines a case not implemented in the code leading to faulty assumptions and not adding extra paths needed. We occasionally refer to the term "real fault", meaning, a fault found in a commercial or industrial system, in contrast to a fictive creation of a fault**.** We define *failure density* as the number of failures found per Kilo Lines of Code (KLOC).

Figure 2.4 shows that we are interested in finding faults (manifested in the code) that propagate (and become visible) at different levels during the testing process. This figure provides a framework, which we can use to relate the efficiency of different TDTs; we can answer questions such as "when is a TDT more efficient?" (i.e. at what level) and "is it possible to find a particular type of fault earlier?"

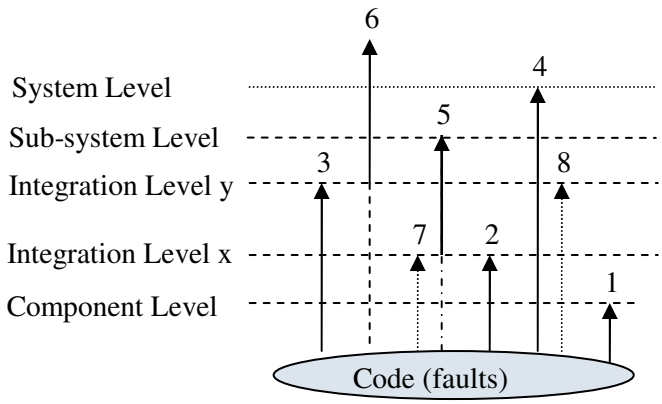


Figure 2.4 Fault propagations to failures, can be captured at different levels

In Figure 2.4, a solid arrow means that the error could be found if a test was entered at this level, e.g. arrow 3, means it can be found at all levels up until Integration level y, where the fault then does not propagate further. A dashed or dotted arrow means that the fault or error is hidden and will not lead to a visible failure (arrow 7 and 8) In Figure 2.4, this means that arrow 1 is a fault that can be found in component test, but does not propagate further. Number 2-8 are errors, since they propagate further in the code, i.e. infecting other parts of the code. Number 4 becomes a failure to the customer, and number 6 has the potential to be a failure at system level – and the customer, if not removed, whereas the others hide *for the moment* in the code. However, they might propagate to a failure if the code is reused or the context changes. Note in particular that the number 6 failure is not visible until sub-system level and even if the fault exists from the beginning, it is not easily found at the component or integration level. Fault number 7 and 8 will never be exposed, since they lie in a location that cannot be triggered in the existing code. All hidden faults and errors are waiting for the right circumstances and context to propagate to a failure. These faults are by Avižienis & Laprie [8] called dormant or residual and are usually possible to find at some level of testing.

2.2 Test Process Introduction

The phases of the process model always exist in the software life cycle, regardless of how fast, or how many iterations you perform within a project. The most basic and fundamental process is the V-model in *Figure 2.5* that is often wrongly viewed as a waterfall model which might have been its original description, but this has more to do with how you choose to interpret it. The W-model, which is a development of the V-model[1] that better captures test design, is presented in Section 2.4 below.

2.3 The Basic Process V-model

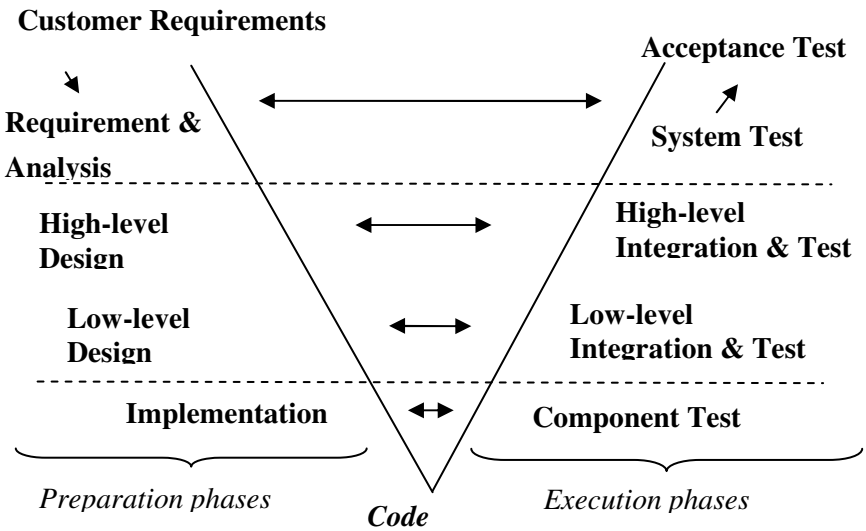


Figure 2.6 V-model – where preparation and execution phases for test are shown

[1] The origin of the V-model is claimed to many. Similarly, the W-model is claimed by many, whereas the true originator is often accredited Paul Herzlich, UK, [87]. We have below adapted the original W-model.

Often names are adapted depending on size (of organisation, system under test) and emphasis. The phases in the V-model are:

Preparation phases:

- Requirement & analysis phase, also including test requirements and test analysis
- Design phase which can be divided into high-level and low-level design, and also test design
- Implementation phase (that sometimes includes component test), including both creation of code, documentation and test code for automation (or test procedures for manual testing)

Test Execution Phases:

- Component test phase (also called unit test)
- Integration phase, which can be divided into high-level and low-level integration, including integration tests
- System test, where tests requirements often are separated in internal phases called function test phases (functional test) and characteristics (measurements of the system) test phase (non-functional tests)
- Acceptance test (for customer release and/or maintenance)

The arrows ←→ are central in the figure and show that the original idea was that each level verified (tested) the corresponding level of specification, and each at a different refinement level. This makes better sense if the left-side is formally defined, and thus the right side is unambiguously a verification method for the formal definition. This aspect is never true in industrial system, since none of the left side properties are open for interpretation, and has several solutions for the implementation in code, and is thus ambiguous to its nature.

2.3.1 Requirement & Analysis Phase

In the Requirement & Analysis phase, the testing is static using review and inspection techniques. There is no hindrance in creating test cases already at this point, based on requirements or other information about the system, but this requires a certain amount of detail, that is seldom visible at this phase. Depending on analysis

techniques used, it might also be possible to derive test cases based on the technique: e.g., using user-scenarios. These can be used directly to define the end-to-end TDTs, creating use-cases in a more formal sense, e.g. using UML-diagrams. In this phase, the test plan is created, defining the set-up of the entire test project. What test plans should contain is well described in IEEE Std. 829 [110], including phases, test criteria, resources, what to test and not to test etc. In particular, in parallel with the system requirement and analysis phases, testers participates e.g. through reviewing the testability of the system requirements. In particular one must gather and state requirements that are necessary for performing adequate test, such as test tool planning, test environment planning etc.

2.3.2 Design Phase

The design phase is where the architecture of the system is designed, and testability must at this point be built into the structure. This is also the time when TDTs should be applied to create test specifications according to the test strategy. At the design phase, where it is possible to use modelling, test cases can automatically be generated from the model or created semi-automatically. There are many ways to fail in architectures, one creating a large “monolithic” system, where components are unclearly defined and the invisible internal dependencies makes maintainability (error-correction), testing and system longevity difficult.
Testing in the implementation phase is crucial in all aspects; this is the time when the code is created, interpretations are manifested etc. We have in [68] given some information on phases of the test automation to take into account. At the implementation phase, static tests in the form of design and model reviews can be performed. The design phase is often divided into “high-level” design and “low-level” design. The system could be seen as a “systems of systems” or a system with many sub-systems interacting. This “software product division” is often related to conceptual, organizational, sellable divisions, as well as pure manageable items. Creating these actual or “fictive” levels is done as a way to manage large complex systems.

2.3.3 Implementation Phase

The implementation phase is where the design is implemented into code, and structured into programs according to the architecture. At this phase, the test teams are also preparing the tests, either by describing how they are manually performed in test procedures or by defining test scripts to be executed by a test tool. Here other quality enhancing techniques like static and dynamic code analysis, e.g. desk checks and code reviews, can be performed. Creating tests cases before the actual code implementation is a design principle referred to as "test-driven development" (TDD) [16]. These are detailed specifications of how to design the software, where software aims to fulfil the specification. For TDD, the tests are actually executable – but will fail, until the code is available to fulfil its intention. This also means a new source of failure is introduced, a faulty test case. TDD is used iteratively during development, but does not represent testing as a measurement of quality, but as a design method especially including refactoring, etc.

2.3.4 Component Test Phase

The execution phases starts when the code exists and together with the component test phase. In TDD the benefit is that the component test phase is hidden in the implementation phase. The test cases are completed before the code is written which is an advantage. Normally, test execution acts like a measurement of completion; when all test cases are passed, the code is "completed". The same happens for TDD, but the amount of test cases is set beforehand cannot be forgotten or skipped when schedules run late (which is probably one of the more common reasons in industry of poor quality). Nevertheless, TDD is mainly positive "normal" test cases that ensures code works "according to intention", and might not have any relations with coverage (if not measured) or good test (if not measured). Therefore, it is a good idea to highlight component test, so that it does not get lost, and set defined targets of what quality should be achieved of the component, and the component in the correct context, regardless of when test cases are written. The component test phase is a crucial step to make sure software parts are reliable. In Chapter 3 the benefit of Software Quality Rank (SQR) is described and how to

get some success with this improvement method for component test. SQR [62][63][67] consists of steps for areas like static analysis and dynamic analysis [9] and dynamic execution in addition to review, but also defines implicit quality improvements like demanding sufficient documentation, automated test suites, and targets coverage criteria [220].

2.3.5 Integration and Integration Test Phase

Integration is done on as many levels as the software is divided in its corresponding design. Applying good integration tests are difficult and require careful analysis. This type of test execution is seldom deliberate, but might be a part of having test cases traversing components and systems, often in longer user scenarios or described as use cases. Here, tests should already have been prepared for execution. One way to minimize the impact of complexity is to integrate and test bottom up, creating many levels of test, and thus making sure each point of integration corresponds to responsibilities, and can be shown to work. Another approach is doing what is referred to as "top-down" integration, thus the system is created (built) and integrated at very regular and frequent intervals. This is also called "daily" or "frequent" builds, or sometimes "big bang tests". It is then possible to minimize late integration problems by performing early integration and daily builds, since large complex systems are then always tested in the right context. The focus here is that if we keep the entire system up and running, the small (one day's change) would in theory make it easier to debug and locate new additional faults, meaning that a consequence is longer fault location for systems with many concurrent changes. Finding a particular fault becomes difficult, since there is no way to tell if it is your change that causes the problem or any of the other changes in the code from the parallel tracks.

The more complex the software is, the more layering is needed for control of the "system of systems", The number of levels of testing in an organization corresponds to both the complexity of the system and the maturity of the test approach. We wish to minimize the time on the critical time path for release, by a massive parallel development. "Divide and conquer" seems to be the best approach when it comes to testing, where every new integration step forms a new system. Stubs

built or simulated facilitate transition into the real integration. It is, of course, possible to view integration problems as an indicator of a series of problems, such as badly defined parameter limits in the interfaces, unclear or dynamic binding of variables, timing and resources issues, dependencies, etc.

2.3.6 System Test Phase & Acceptance Test Phase

In these phases, the entire system is tested, including both functional and non-functional aspects. The difference between the system test phase and the acceptance test phase is in the goal of the testing and the depth of testing. In each test execution phase, any group of test cases can be performed: e.g. functional and non-functional tests. These types of TDTs (See Chapter 13 for detailed descriptions) should already be thought of during test case construction, and planned for in the test analysis phase. At all above test levels, test results are collected, analyzed and reported, which serves as a measurement of the quality of the software, and also aids in improving the quality by targeting areas to correct.

2.3.7 Advantages and Disadvantages with the V-model

The advantage with the V-level is that it is simple and clarifies both that testing takes effort, the concept of levels, and is at the same time explaining that verification is taking place at the same level – and with a specification from the "left side". This is what the double-edged arrows in Figure 2.6 mean. This view using the V-model of software test in the development process is one of the reasons test maturity seems to remain low in industry. Instead, test should be in focus from the beginning of the development. It is even suggested that the requirements are captured together with a tester at the customer site, in the spirit of making sure that the requirements becomes measurable and testable – and to the point. This could also be implemented in another way, by making sure a representative of the customer is a part of the development (and test) project. It is often complained that most existing models are insufficient due to their

waterfall nature. In our current thinking, the ordering of these phases is natural and always included. We note that sometimes these phases might not be documented, or could be performed very fast and not thoroughly thought through. It is hard to build a system without any requirements. It is hard to demonstrate the code execution – without the code being implemented and executable. The V-model should be interpreted as an iterative model, and not a waterfall model.

2.4 W-model

In the V- model the early test phases are hidden and not highlighted in the process, which makes it easy to ignore them by not providing sufficient resources and time. In our description, we have incorporated some of these aspects. In Figure 2.7, the adapted W-model based on Herzlich [87] the test effort (and consequently the rework done by development in the test execution phases) are much better highlighted. The W-model clearly separates the requirement phase and the analysis phase, even if both of them often occur in parallel and interact with each other.

The early test phases added here in the W-model (Figure 2.7) are:
The preparation phases:

- Test Requirements
- Test Analysis
- Test Design
- Test Implementation
- Test (environment) Preparation

The new Project phases:

- Re-work

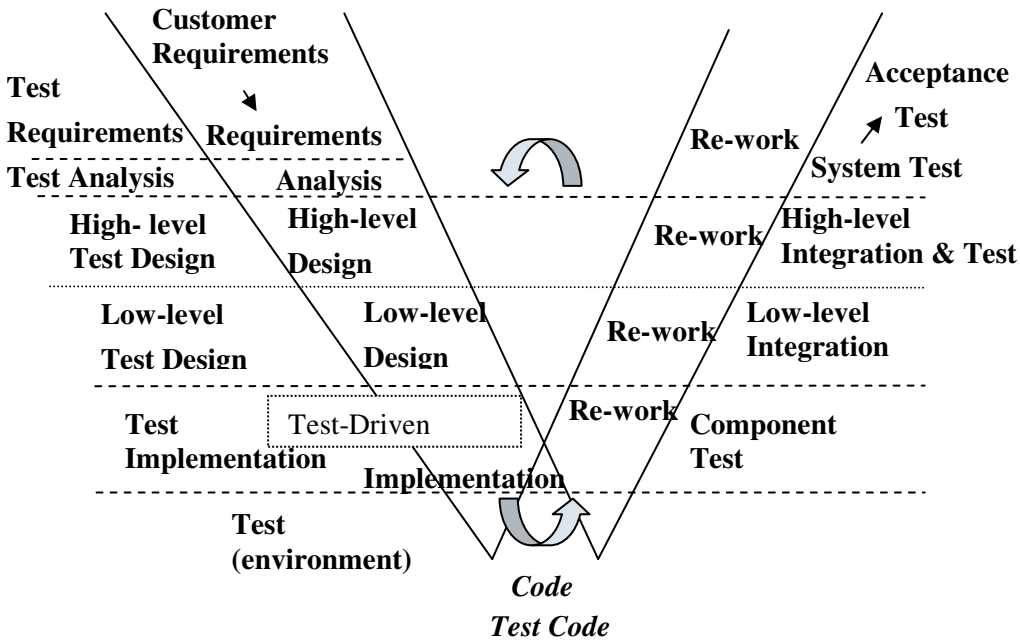


Figure 2.7 The W-model adapted to industrial parallel & iterative/incremental design

2.4.1 Test Requirements

Test Requirements are unique requirements outside the system requirements, such as defining tools, test environment, and other constraints that impact the software test. See more about this in the presentation of the V-model in Section 2.3.

2.4.2 Test Analysis

The most important phase is the test analysis that contains a review of the requirements of the system and software to be tested. This includes defining the scope of the entire test and then limiting expensive and time-consuming testing which is increasing the risk. Defining the test goals and understanding constraints set by the system, software, testability and development is important at this time in a development project.

2.4.3 Test Design

Test Design means selecting the techniques, approaches and methods for implementation. What should be done manually, what should be automated, should we automate the implementation of the test case, or only the execution? This defines how the entire execution is to be done. This phase is the main focus for our research. The high-level test design should also include creating architecture for the test cases (and test systems, test approaches, etc.). The division of the specification into high-level and low-level is often a reflection of the system complexity and partitioning. Thus, for large complex systems, an entire department of testers can be designated to focus only on one aspect of the test, e.g. testing the performance under special conditions. It is no use in doing this only for parts of the system, but should be done as a final effort, to get adequate measurements. Another group or department could show the stability and robustness, and a third group test the functionality of the legacy aspects of a system.
The more complex software is and the higher the quality requirement is, there is typically an increased division of focus in testing and specialist roles for different types of testers. This must be taken into account when doing test design. This thesis has not focused on all aspects of test design and approaches, as earlier described. The result of test design is often the test specification, including specific documents such as test environment specification and test tool specifications. In addition one may need to define specifications on data depending on the context of the test and the domain of the system being tested.

2.4.4 Test Implementation

Test implementation means applying the test design to create an instruction or automation of a specific execution of the system. The implemented test case contains enough information to execute the specific system from a specified point. The result of test implementation (from conceptual to an explicit test case) is either text or code. In IEEE Std. 829 [110] from 1998, the textual description is called the test procedure and its corresponding code for automating the test procedure is called a test script. Both are in a sense the

implemented test case. A series of test scripts is called a test suite. It is often the case that only manual test cases need a textual entry, otherwise it is simpler to go directly from the test specification to the test case script. The test case script, or test code should have sufficient commenting, especially about expected input data (and expected output), dependencies etc. A sufficient description could be kept as header information in the test code. Note that some "test specification" information on a higher level is often beneficial. It is e.g. easy to forget assumptions, goals, dependencies and traceability items; the latter should also be mirrored in the actual test case or easily linked and found. One of the problems is keeping up with the changes and version-handling of test code, since test cases evolve, but should still be kept to work for each version of the software. In practice, test code should be treated and viewed upon in the same manner as the code, since it is a similar asset, with similar importance for industrial software.

2.4.5 Test (Environment) Preparation

The Test Preparation phase contains setting up the specific context and environment for testing. For most industrial work this is a very challenging task. Different types of real scenarios need to be created and mimicked, often in conjunction with specific hardware. Other activities are setting up tools, preparing data in a "test" database, and creating simulators and emulators. In principle, this phase can be done at any time after the test analysis is complete and when the test design has defined what and how to test. Test environments are an important part of the test requirements. Often, when test is finding failures, the reason can be traced to an inadequate test environment. The test environment sometimes needs to be a complete replica of the real environment, e.g. when testing space or medical equipment.

2.4.6 Re-work

Re-work is stated on the "design" side and is needed on all levels, and has the generic meaning of representing the same action as on its left side, but the focus is on locating faults and correcting them, which also invokes rewriting documentation, specifications and many other

aspects of development. Since it is not easy to develop fault free software, several iterations of refinement and quality improvements are necessary. The amount of re-work is frequently underestimated, causing a delay in releasing a system with the required quality, if adequate resources are not sufficiently planned for. Tool support for fast correction of faults speeds up this re-work phase. These activities are well highlighted in the W-model.

2.4.7 Newer Test Process Views - Test Driven Design

Software development processes are continuously improved and changed, to challenge, change and make people motivated. Many "new" testing trends are pre-dominant in industry today. Simple test methods and approaches prevail, and these do not change fast. In many systems, emphasis is spent on agile, fast and lightweight processes. Some of these processes aim to minimize the testing effort, which often implies to eliminate a formal or structured approach that requires detailed specifications. These new processes also re-order when and how to test. Using test cases as formal low-level design specifications in a very iterative approach, including describing the test of fulfillment, implementing the code, and then re-factor the code is a part of Test Driven Design, TDD [17]. This seems to boost the view of how test cases can be used for developers. Some "unnecessary" tests will be created, since software development is an iterative and creative process, and faults during intermediate steps will also result in test cases. Well-performed TDD definitely improves the developers initial quality due to the massive know-how of testing that is needed to perform this type of development. Limitations with TDD is that the test cases are in nature focused on "making code work", instead of testing to find faults (see later discussions about positive and negative test in Chapter 9 and 13). Using TDD does not take away the need for thorough testing after development is complete.

New "Agile" processes, "Scrum" or similar processes, often ignore the fact that independent testing is still needed, or do not put enough focus on it, even suggesting the testers role might diminish or even endanger the tester's role during organizational transition to agile

[35]. Ignoring substantial testing after a system's components are merged into a complex system is risky and may result in poor quality of the system. This does not mean that the ambition of TDD (Test Driven Development) is in anyway wrong, since all ambitions making sure that developers improve their own testing, will be improvement to quality. This is usually based on the simple fact that doing the test cases first will under industrial time pressure mean that test are not skipped in the last minute, since it is easier to skip a test, than completing the code. It is unfortunately very risky to not spend time evaluating the coverage and do other quality improvement tasks, e.g. refactoring, to improve the code quality after the first attempts.

2.5 The Plethora of Publications in Software Test and Test Design

The area of test design and TDTs has been a focus of publication in software testing for more than 30 years. Juristo et al. [128] describes an overview of 25 years of testing (2004), and even if very selective, gives an insight of an area lacking substantial research. Even if the number of books and articles are continually increasing, the area have instead of making clarifications, been drowned by terminology and interpretation problems that relate to the different systems under test, the human innovation, and the need to avoid using the same names dues to lack of knowledge, and – if knowledge exists – based on the fact that copyright laws prevents definitions to remain exactly the same. The need to sell "old" as new with different names, context and focus becomes a way to "renew" the area. Instead of making it easier to comprehend, we get a dilution of the content. This makes TDTs, and test design particularly difficult – since the exact interpretations and definitions are often lacking, but assumed – and can vary as much as there are different techniques. Examples of this can be found in the following books:[4][25][26][34][42][43][78][87][96][99][135][127] [132][189][204][198].

There are few researchers who have aimed at making order in this plethora. However, this requires extensive know-how of actual testing, which is rare in many academic institutions. The focus is often on one technique or approach that is compared with either random or an "as is" (unmeasured) test suite. It is often not feasible to

explore series of techniques, and many of them seem beyond the time-limits or scope-limits of a PhD. Unfortunately, the lack of deep know-how results in a "new" technique being considered as an original work, even if it in many aspects is "exactly" the same as an existing technique, in the sense that it results in the same test cases. Instead an identical or variant of an existing technique has been created, maybe only different in representation, style or human involvement.

TDTs were first mentioned by Myers [163]. The seminal structure is presented in the book "Software Test Techniques" by Boris Beizer [18], although he is not the sole originator of these techniques. A novel attempt to define negative testing techniques can be found (based on usage of systems) in Whittaker [211]. We have dedicated Chapter 9 to sort out the negative usage approach from traditional view of TDTs. Unfortunately, sorting out seminal work for all techniques is a PhD in itself.

In modern times, few researchers have attempted to make order in this plethora of TDTs, and what stands out are in particular Vegas [203] work, that we will partly contrast ourselves to in Chapter 13. Murnane's [157][158][159] work, has been instrumental to view the techniques in a different light, thus clear definitions of them is a problem. Furthermore there are many research works for specific groups, e.g., recent work from McMinn [150] with the search-based testing techniques as a specific focus or Jia et. al. on mutation testing [124]. Other relevant works relating to each of the studies are discussed is each chapter/study, respectively.

2.6 Historic Classifications

The most commonly used differentiator of TDTs that seems to be dividing techniques into black-box and white-box testing. These concepts predate software testing, and black-box has a common interpretation of not looking inside "a box" but merely observing input and output behavior. White-box was intended to be the opposite, full access to whatever is "inside" the box.

One of the more influential books about Test Techniques (TDTs) is by Boris Beizer [18] and his follow-up book is named just "Black-box techniques" [21]. "White-box" became figuratively speaking the name for using the code itself for the technique. Since you cannot see

through a white-box either, the term glass-box and clear-box were coined. And “new” meanings became attached. The glass-box gives the possibility to view the inside, but no possibility to change the software (often the case with third party software, where support-licenses cease if code is tampered with in any way), and clear-box – which would be the full access to the code in all aspects.

The problem with black-box and white-box used in the context of TDTs is that the meanings changed over time, ever so slightly: Black-box became techniques, where only input and output behavior was interesting (often the subtext without regard for how it has been implemented). The most common interpretation of white-box techniques became synonymous with code coverage techniques.

Hence, this was often interpreted as white-box testing (TDTs) are TDTs used by developers (since they also possess insight to their software code, structure etc) and black-box are TDTs that is only concerned with 1) input and 2) targeted at testers which assumed to have no insight – or should not have insight in the implementation when designing their test cases.

Some would go so far and read “all goals of testing” into black-box testing, which is of course skewing the initial message. This assigned meaning has unfortunate impacts, and created a testing approach that is ignorant of internal structure, and developers that bother less about behavior. It is possible that this fuels the popularity of development and test where people work together, and requiring both roles to have as good understanding of both aspects of software.

Developers need also to test input-and output behavior of their code, which makes every object a system in it-self, useful for “black-box tests”, and every tester can test better knowing structure and implementation of the system – as well as complementing with different types of coverage. This we could conclude as a result in two of our studies, both Study 1 in Chapter 3, and Study 8 in Chapter 10.

Since these concepts originally came from an observation viewpoint, what could be observed of the software (or hardware). It is imperative that one should understand what is “inside” the box, even if the reachability is through the interface, since the intention is to make better test cases. Taking this viewpoint one step further – test should propose requirements on internal states, values, and parameters to improve the testability of the interface.

Due to this unfortunate use of the original concept, it is not fruitful to linger to them. This is also concluded in a book by Ammann and Offutt [4] who are referring to these concepts as “old-fashioned”.

Therefore newer approaches are better – since they disregard role and level, but focus on the concept of the technique. This would then be “functional” with the subgroups “input-related” and “path-related (structural)” and “functional” vs. “non-functional”, which are views, proposes an entire new type of organization of the techniques. We can already see this view separating, and people want to bring in “experience based” techniques. Since these are un-measurable, undefined and largely ad hoc, our best guess is that this is a mix with usage techniques.

In Chapter 2.5 below we propose a structure for TDTs as a starting position for some of the more well-known techniques. Our selection has a personal bias and in no way intended to be complete in the plethora of techniques that exists.

Many of these test approaches can be used at any level of testing, but does not have the same *strength* and *purpose* at all levels. Structure aims to define some form of “order”, structure that can be either “graphed” or “path” or countable in a linear fashion or parallel (linear) in contrast to lack of order, often random or ad hoc. Some confuse structural test to only be looking at the code structure.

Structural test it is not solely defined as path, which is easy to assume. Structural could also be defined as anatomy (architectural hierarchy), or any type of order. E.g. every menu item, every GUI-windows, would be a possible structure. Other examples of structures are in relation to a specific order of tasks in a process. This definition of structural test is especially useful when testing parallel executions, where we must differentiate the actual execution from the code and from the system usage. Testing the behavior (functional testing) is possible whilst doing it in a structural fashion, and should be kept in mind when approaching fulfillment of e.g. coverage goals. These simple definitions already have a series of disruptions, when a standard refers a characteristic as a “functional characteristic”, e.g. functional security [117], mixing the functional aspects with the non-functional.

2.7 Overview of Test Design Techniques based on Groups

This structure was created based on a biased selection on a series of documents, trying to honoring some of the current basic definitions, and still preserving our end result. As we will see in Chapter 13, we will collapse this structure in a better taxonomy in the form of "families", and hopefully better define overlaps (subsumes, includes) and variants.

A. Functional TDTs:

I. Structural techniques
    a. Coverage-based
        i. Control flow
            1. Functional block
            2. Statement block
            3. Statements
            4. Branch/Decision
            5. Weak condition
            6. Multiple Condition Decision Coverage (MCDC)
            7. Strong condition, Branch Condition Combination (BCC)
            8. Linear Sequence code and jump (loop) (0, 1, all) LSCAJ
        ii. Data flow
            1. All-paths
            2. All-predicate uses
            3. All-definition uses
            4. All-uses
            5. All-pair uses
            6. (and many more….)
    b. Order of use
        i. Integration based
            1. Top-down
            2. Bottom-up
            3. Big-bang (all)
            4. Incremental or continual
            5. Mock/skeleton/wrapper
        ii. Usage based

1. Anatomy
2. Development (availability)
3. Testability
4. GUI/menu
5. Process/task

c. State- transition/State-chart
   i. 0-switch
   ii. 1-switch
   iii. n-switch

d. Models (including c)
   i. Use-cases (UML or not)
   ii. User scenario's
   iii. All objects/items
   iv. Classification Tree Execution (CTE) could be viewed as combination technique
   v. Cause-effect graphing
   vi. Property based (in combinations with IIIe)

e. Mutation Test (also as own group in IV)
   i. Syntax Test
   ii. 1st, 2nd, n-and High-order mutation
   iii. (and many more….)

II. Input Domain Combination Analysis
   a. Full matrix
   b. Call-pair, Pair-wise

III. Non-structural techniques (execution path related)
   a. Random execution (path)
   b. Random permutation (of existing paths)
   c. (and many more….)

IV. Non-structural techniques, Input-related techniques such as:
   a. Random (selection from input domain)
   b. Fast Anti- Random (FAR)
   c. Specification based, positive (normal) input
   d. Dependency/Slicing
   e. Negative input techniques
      i. Outside of boundary
      ii. Too large, too small, too short, too long
      iii. Misuse type, character-sets
      iv. Magic set (0, Empty set, null, - negative, float)

v. Damage file/media
f. Equivalence partitioning (Category Partitioning)
g. Boundary value(s)
h. Computational techniques
i. Random
ii. Statistical selection
i. Search-related techniques
i. Hill-climbing
ii. Simulated Annealing
iii. Evolutionary/genetic
iv. Ant-hill optimization
v. Tabu-search
vi. Cross-Over, substitutions
vii. Multi-objective
viii. (and many more….)

V. "Other groupings" with attempts to select functional test cases:
a. Ad-hoc "experience" techniques
i. Error-guessing
ii. Experience based
iii. Value-based
iv. Risk-based
v. Priority based
vi. Exploratory
b. Usage based
i. Statistical usage test
ii. Usage profile
iii. Repetitive
iv. Exploratory
c. Fault-related evaluations
i. Fault-injection/fault seeding
ii. Mutation testing

B. Non-functional TDTs:

I. The "abilities" based on ISO/IEC Std 9126[119], ISO/IEC 25000[118] Quality Model
a. Functionality set of properties
i. Suitability
ii. Accuracy
iii. Interoperability
iv. Security

v. Functionality Compliance

b. Reliability
   i. Maturity
   ii. Fault Tolerance
   iii. Recoverability
   iv. Reliability Compliance

c. Usability
   i. Understandability
   ii. Learnability
   iii. Operability
   iv. Attractiveness
   v. Usability Compliance

d. Efficiency
   i. Time behavior
   ii. Resource Utilization
   iii. Efficiency Compliance

e. Maintainability
   i. Analyzability
   ii. Changeability
   iii. Stability
   iv. Testability
   v. Maintainability Compliance

f. Portability
   i. Adaptability
   ii. Installability
   iii. Co-existence
   iv. Replaceability
   v. Portability Compliance

II. Characteristics (aspects) /should probably be included as headings in I.
   a. Robustness: Stability, Durability (as part of Reliability)
   b. Vulnerability (as part of Security)
   c. Recovery and degradation (as a part of Changeability and Timing behavior)

III. Performance
   a. Throughput/response time
      i. Individual task
      ii. Series of tasks
   b. In normal, minimal and stress conditions:
      i. Random overflow

ii. Load /memory/transactions/…
iii. Volume test
iv. Number of users
v. Number of computers
vi. Number of entities (e.g. parts of database stores)

C. Combinatory TDTs (2 or more techniques) Input variations: e.g. Equivalence Partitioning + Random Input
a. Fault Injection + Search Based Testing
Etc.

The above techniques could be discussed at length, e.g., regarding grouping, selection technique, value of contribution, actual clearly defined origin (or lack thereof). The purpose of the previous list is to give an initial overview of the existing plethora, and as an attempt to narrow this area down to a mere few, distinct and useful techniques, thus we have omitted a large selection of very specific techniques, detailed variation etc. In fact, each of these techniques or sub-group of techniques have earned people PhD's but too few researchers have attempted to do related research amongst groups, and this was one of our motivations for our set of studies; to really establish how this comparison could be improved upon.

Part II
Empirical Studies

Chapter 3. Component Test Improvement through Software Quality Rank

3.1 Summary

At Ericsson, we defined a step-wise improvement model, called Software Quality Rank (SQR), to guide our design teams in component test improvements (see [62][192] for more details). We have piloted this model during an 18 month project, commencing 2003, where 9 design organizations with a total of 22 design teams from around the world participated. Now SQR is used both across Ericsson and at other companies, contributing to better quality. The experiences while applying these experiments and the results of the experiments, have given us insights in difficulties of deploying quality improvement programs.

What is Software Quality Rank? It is an improvement program focused on the definition and improvement of component test for software developers. The basis of SQR is the idea that there exists a way to select what parts of the software that should be improved, and if a strategic choice of what components to improve is made, it will impact the overall system quality. SQR will take into account new code, legacy code and modified code. There is no secret that we are inspired by Software Engineering Institute's Capability Maturity Model (CMM) [175] [103] in the way to stage this as 5 levels. The main difference is that CMM addressed the whole process whereas our focus is on component testing alone. The old Swedish school-system had a 5 level grading, where 3 means "pass", which made us – in contrast to CMM, not thinking the ultimate goal is that all code achieve level 5, but that the majority of the code should instead achieve "pass".

3.1.1 Context of this Study

This initial Study served as a part of the main motivation for this PhD. The setting is a typical improvement project as conducted in industry. The main goal was to improve the quality at the code phase where bugs are manifested, thus by impacting the developers and software designers. This study triggered the interest in better understand that test has a direct relation to a quality system, and the strong power of using coverage [220] at the development level.

3.1.2 Design of Study - Research Method

The research method in this Study could be compared to an action research project, where the researcher participated and influenced the studied case to evaluate the change. Lewin's [143] original model from 1946, of action research included iteration of six phased stages, in which we also explain how we approached our improvements:

a. **Analysis:** Our basic theory was - the more code that is executed the more failures would be encountered, and that statement coverage was a basic very powerful test technique in conjunction with using static analysis tools and code reviews. Our analysis was based on measuring out statement coverage as an initial step. Our first work was for some design groups to make sure they had a test environment that was feasible, e.g. tools for measuring coverage.
b. **Fact finding:** We found several research based papers on the concept of improving the quality, use of static analysis tool [72], and measuring coverage [220]. Reviews are well documented since Fagan [75] – to improve code quality, which was also used extensively through the *Cleanroom* movement [151]. Improvements are often deploying by setting up a baseline and then defining "steps" of improvement, see CMM or CMMI [175].
c. **Conceptualization:** Defining the steps, and the content of the SQR model. Based on our experience with CMM, and step-wise improvement, we defined 5 steps, where the average would be modeled after a exemplary developers quality approach. Then we targeted Rank 5 for the top quality code, in our minds that should be safety critical code. Then rank 1 should be baseline – and the actual "ranking" for the improvement steps. 2 and 4 came as

natural steps in the scheme after this. These ranks were then reviewed by a small hand-picked group of developers, testers, quality improvement people for better descriptions and refinements.

d. **Planning:** Targeting and setting up the criteria for the improvement in practice for one entire industrial project. Meaning – defining the goals for the projects, such as acquiring tools, and then teaching the methods and tools. In particular, meetings were set up before, during and after to follow up. But particularly following the project planning.
e. **Implementation of action:** Deploying the SQR model – and collecting measurements and observations, by creating template and, interviews. Two Master Thesis Students [192] aided in acquiring and understanding the set up.
f. **Evaluation:** Assessing results and the lessons learned from the study are tasks performed as conclusion/evaluation. In particularly the number of failures found was assessed as well as the improvement in coverage, test time for the next level, and number of test cases. We then reviewed our results as a post processing, at a much later actual time, to give the actual system under test time to reach the field, and get a more honest feedback, which in this case strengthened our results.

The main problem with replication of this study is the inability to publish the underlying data due to organizational policies. Since the researcher was instrumental in the entire study, in all stages of defining and deploying – there could be a strong researcher bias in the achieved results. Success might have been entirely person-dependent. The measurements were gathered independently, with no researcher impact on the result – which at least gives some confidence in the results. Since the method was performed by design teams from different select countries, but not a randomized sample – and all design teams from the same company, will also indicate that the result has a bias that compromise the results. Since this software is embedded together with a lot of other software – it is impossible for a customer to deliberately manipulate the result or impact the evaluation. Another threat to data validity is that some of the data collected was in the form of a report, whereas log-files were not always submitted, and hence the possibility to tamper with the result

cannot be ruled out. Overall, one could only claim this is a typical industrial software concept, and gives a good indication.

3.1.3 Contribution

The main contribution of this study is establishing a ranking method to achieve software quality improvement, and deploying it in many design teams, achieving a successful (for the industry) result. The contribution from research perspective was gathering observations on the problems and usage of an improvement model, establishing factors that seem to contribute to the success (improved product) by the use of coverage, static analysis and reviews as a quality enhancement tools, and understanding limitations with these approaches. Finally it became clear that 100% statement coverage does not always means better tested than, per se an 85% statement coverage, but that other factors impact the quality results, such as know-how of the system that makes you select good data in combination with an enhanced coverage at the right places. A contribution is also realizing that improvements of the component test has a great impact on the overall robustness of the system, and impacts the software quality and all consecutive phases. And that even a small but well selected part of the system can make a big difference in how the system is viewed as a quality system.

3.2 Software Quality Rank

This improvement program is based on the assumption that some form of component testing is already done, since design organizations have delivered software that has been used commercially for years. The problem we are aiming to address is that too many failures are found too late in the different testing phases before delivery, which makes development costly, and that component test is therefore an economical target. Testing quality into the system at the functional or system test level is possible, but the result is long delivery times since regression tests need to be performed in addition to a lot of failure administration (e.g. analysis, debugging, correction, and fault prioritization and handling).

The idea is to move some of the test effort from the test organization to the development and design organization. The extra time should then be spent on quality enhancement at component test, before handing over to the next test phases, such as functional and system test. We wanted the quality to be more predictable at an earlier stage, so that we could estimate better how much testing remained before release. We analyzed our failures and faults, and draw the conclusion that many of these faults could have been prevented at much earlier stages, when they could have been manifested. It seems logical that the lack of test education in the design organization has an impact on quality. Most universities lack education in software testing, except for brief introductions as part of software engineering courses. This is not sufficient knowledge required to perform solid testing. The most common cause to bad quality from developers is that there is not enough time in the development phase set aside for quality improvements.

We believe that by introducing a special improvement program we can support developers on what quality work they should focus on, [62]. What is more important is that the selection of targeted components has to be done by the developers themselves. This makes the developers committed; since their choice is based on what they believe is reasonable to perform within the time given. When developers have an efficient test environment, good tools at hand and the knowledge of how to produce quality software, they are more likely to do so. They become more "fault aware" and we can also see a trend that they mature into writing and designing software that is more testable. All of this results in better code and quality software. Software Quality Rank consists of five improvement steps that we have attached with different associated meanings.

3.2.1 First Rank – Quality Awareness

In this phase, you select the code to be targeted for improvement. If you have legacy code, you must select which of these components that could be targeted for improvement. This means that you have to be "quality aware", or in other words, know what quality your entire software has. This awareness will make it possible to focus your efforts in a cost efficient way. You select targets by a series of actions. We suggest that you investigate what tools you have at your

use, or can get to work in your environment. With these tools, you collect basic measurements on your code. We suggest simple measurements such as Lines of Source Code (without comments), what your current test suite yields in coverage, any complexity measurements that could contribute to understanding the software, for example, call-pair, nestled calls, McCabe's Cyclomatic Complexity [146][148], or Halstead Volume [91] metric. All of these measurements are just contributing factors to support the judgement of selecting improvement targets, but should not be defining.

The most important measurement is to use known failure statistics of the different components and try to establish the number of faults for each component. All the software that is in production, in customer use, and performs well should be "removed" from the target list. With performing well, we mean components that have none or very few and seldom reported failures (anomalies) or of too low severity. The target list now only contains candidates for improvement, but they should be prioritized, from the worst components (many high severity failures) to bad (some failures) in a strict ranking order. To this list you should now apply business aspects, which imply that you remove short-lived components, low-usage, low market value etc. What remains now, is a list of the possible targets among legacy components, where your developers agree these are all the main targets of improvement. You could also add subjective aspects of selection, e.g. components with bad design, component code difficult to maintain, written by many different developers etc. At AT& T a tool [174] has been created to help in targeting the 15 % of the most what they judge as quality targets. The next part of creating the targets is to prioritize all new code. The reason is that they have not been tested and we can assume that they are more fault-prone than legacy code. Depending on the amount of new code, which is often substantial, our observation is there will not be enough time to target all new code created.

We suggest you to prioritise also the new code based on importance. Finally, the new code is impacting the existing code – which we call modification code, should be selected on a case by case basis. Sometimes you change very little, but in an intricate and sensitive part of the software. Even minor faults in these parts could propagate to severe failures. Sometimes a "small" change affects a lot of code (e.g. 5 % change in a lot of software), and then selecting one part as target might not be simple.

These three “lists” of legacy, new and modified target components should now become one list, with an emphasis on new code, some really bad legacy code, and the most important modifications. If any of the lists coincide, for example, if the new code will impact the legacy in a bad place and modifications are risky – this could be a target. The resulting list should be what the developers themselves believe are the most important areas to spend additional time to improve the quality on. This is the “baseline” of what we suggest should be targeted in the next project for special quality improvement. In a time to market software development, what *should be targeted* for improvement, will probably take too long for the projects time limits. Therefore, it is important to do a selection from the top of this list, within time and budget limitations and document that in the component test plan for the project. The selection should also consider practical aspects, for example that all developers have at least one selected component each, to achieve team improvement.

This **component test plan** is a key document for SQR. It is important for the long and short term (project) view that defines what should be targets for quality improvements. It will work as a suggestion from development to management of what should be targeted, and if management does not agree with the limitations of the list, more time should be added to development, or fewer components could have that extra quality attention. The plan will also serve as a “contract” and will be commitment from the developers, and a good aid for management to follow up progress.

The final requirement in this quality awareness phase is to investigate the possibility to perform the next steps of improvements. This means that tools to measure coverage must exist that can handle automated test suites and a test environment that works with these tools must be created. If no tools exist, they must be procured, installed and sufficient training should be provided as a part of the component test plan. The last task is to define a checklist adapted to the own tools, process and terminology. This checklist is a way to assure that the selected components have reached their SQR level. Each improvement requirement for each rank can be transformed into a question that will be checked before delivery. These checklists make it possible to follow up and hand-over correct information to stakeholders, such as project management, test organisations and line management.

3.2.2 Second Rank – Quality Improvement

In rank two, the actual improvement is performed on the selected targets from rank 1, as defined in the component test plan. Rank 2 will define requirements for these improvements that is often listed, and checked against. Reviews are common practice, since many decades, in Ericsson, and every project defines which documents should be reviewed. We believe that code reviews and quality improvement reviews are often dismissed by development projects, which usually spend their effort reviewing design specifications and requirements to make sure they are clear and understood. In rank two there is a requirement that code should never be a *one-person responsibility*. It is important that more than one person have reviewed the code. We do not believe it is cost efficient to review all code with the entire design team, but it is important that selected parts of code should be reviewed. In particular, we suggest that header files should be reviewed, since they specify interfaces, parameters of importance and other valuable information. We have the requirement that input value boundaries (maximal and minimal values) should be explicitly mentioned in the header files, to ease the creation of test cases. In addition to the above focus, the review should also include if the code is effective (in terms of performance, memory consumptions etc) and if there are special dependencies to other components that matters.

This is also the place to measure the percentage of code comments, where 0% is not acceptable. Estimating what number of code comments is acceptable, depends greatly on its content. The quality (content) of the comments are more important than the number, which means that comments are reviewed with the aspect of maintainability and usability for another developer, which currently is only a judgment.

It is difficult to define the minimal documentation for a component that would enhance understanding and transferability. We suggest the best approach is to try and capture what a developer would tell a fellow developer, to add just the right information that minimizes the time to read and understand the code. We suggest that the easiest way is to use modern tools, basically have the main designer explains the code, and either make a video-clip of this information into the software repository or take a photo of the sketch and make that a part of the documentation. Time should not be spent on poor drawings. If

any tool should be used here – it should be more of a modeling tool, which can provide other benefits, in addition to the

Other targets for review are the automatic test scripts, since we have noticed that this code is often at a much lower quality, containing hard-coded values instead of a maintainable test-script.

We suggest that simple measurements from the reviews should be collected at this stage. Suitable measurements could be, i.e. number of participants, time of preparation and review, and the number and severity of faults and improvements. These simple review metrics will not require much extra effort, but will give the group an indirect way to evaluate their efforts in review, and stay focused.

Assuming that the code is prepared, we have a requirement that static analysis tools should be used. The SQR "categories" was initially used only to categorize different warnings from a Static Analysis tool in five different stages. The reason being that there are a lot of warnings and it is not easy to judge the importance of the warnings. We do not want to over-exaggerate the contribution of such tools, but they are definitely helpful. The aim is to make developers see these tools as an extra pair of reviewing eyes, and thus as an aid in their task to desk-check their own code, instead of a burden of abundant set of warnings. The aim is to execute the code with the tool, analyze the result, and correct as much as possible. If warnings remain in the code, any new warnings are easy to miss. We think that the tool Coverity have brought this to a new level, finding problems rather efficiently. We of course recognize that there are a lot of different static analysis tools with different advantages and disadvantages, e.g. Parasoft, Lint, Flexlint and similar [72].

In addition to reviews and static analysis tools, the main task of this phase is to make a sincere test improvement effort. This means that we teach testing techniques [182], such as equivalence partitioning (EP), boundary value analysis (BVA), state-transition testing (ST) and the different coverage measurements as structural techniques. The requirement is that at least equivalence testing is performed, with exercising both allowed and disallowed parameters (the latter we call "negative testing"). We have noticed that many developers in large complex systems tend to execute their load module in the existing context of the system, instead of spending their time to stub every aspect. They are also primarily using functional (traditional "black-

box") test approach, where they are exercising the normal case of the component through its interface. Here the aim is to make developers aware of the limitations of such testing, with the first goal to create many more functional test cases that would be as complete as possible from a functional point of view.

We measured the number of test cases that existed, and how many new test cases have been produced, and did not pay attention to how big a test case is. We have the requirement that normal cases and the most common fault cases should be executed. We encourage an "automatic regression suite" of the component to be created by programming test scripts. These test scripts should be treated as normal code, have header information etc. The test scripts do not need extra written documentation other than a very high-level test specification. The test script should clearly specify what it tests. There is no need to say how much or what should be automated. We assume that developers like writing code, and this is a natural way to create tests. What might be new is using a test harness tool, or a test framework with templates available, which will ease the maintainability of such test scripts.

These test suites are then measured with a coverage tool, which should give the developer an adequate feedback on how well it is tested. We have used the general assumption that 50-70% statement coverage means the normal cases of the code have been covered. To test fault cases, you have to add more test cases. Using coverage to create test cases is a good help to make sure the code is understood.

We encourage 100% feasible statement coverage [220], which gives room to decide what is feasible (economical, cost-efficient, possible) to perform. A component can sometimes consist of several hundreds of files. A deliberate priority within the component should be done on what files that should achieve 100% statement coverage and which should not. The average of the component could be as low as 85%, since good code in our context is assumed to consist of many security and safety entries that can never be reached. Also we have noticed that software that handles hardware might sometimes be harder to test in full (as well as kernel code of operating systems). 100% statement coverage does not mean that it is completely tested. At the end, it is good testing we want, and not a good measurement. Yet, the developers have to be able in an assessment to justify their achieved

coverage and explain any low numbers. We have seen the coverage to be a very beneficial tool, if used with sense, and in the order we have suggested.

Finally the rank 2 requires that memory checking is performed using tools, such as Purify [107]. We have also an option that profiling can be used if it is applicable at this low level. Having an efficiency check of the code is useful, since many small components contribute to the overall performance. It is a good stage to capture performance problems. All these items are then checked, and documented with appropriate logs and references in the checklist that is delivered with the code.

3.2.3 Third Rank – Transfer Quality

The third rank is aimed to be the goal for most components (95%) in commercial software. This rank level is based on what senior developers, with good quality sense are doing to make sure that the code and its documentation is sufficient, the code is possible to transfer, and the code is maintainable without extra investments. The idea is that the improvement here becomes only some direct actual doing, but more of a checking of the component to make sure all documents (incl. test docs) are in place. The reviews performed should be with the additional focus that the documents are good enough to "handover" to another party, and that review meeting should be with stakeholder's presence. Here, testers can be invited to review test scripts and test specifications, and a maintenance organization can participate in both design and code review for selected parts of the software. This could be planned from the beginning of the project, and rank three should not be a costly phase to achieve.

The focus is again to improve test by adding tests – by exploring the input better, i.e. making a boundary value testing (three values for each boundary). Also loops, nestled calls, implicit else etc should be explored. The aim is improve the testing with more and better fault scenarios, and the goal is to reach 80% feasible branch coverage, and explore basic conditions if they are prioritized. Parts of the code could be target for state transitions or state chart testing. Initially we had several measurements (complexity) here, but this has been dropped,

since we feel they do not contribute to quality improvements. The aim is to conclude that the component has been tested, measured, reviewed and have sufficient documentation to be transferable with a small cost. In rank 2 we believe the component is ok, where as in rank 3 we are confident will perform ok. Yet it is important to point out that this is a cost efficient judgment on the component, and we have made an effort to find the "right" level.

3.2.4 Fourth Rank – Critical Code Quality

At rank 4, we change the concept from discussing components to discussing code. In particular, we are selecting critical or central parts of the code, within a component that should be of rank 3. This could also be code that should be optimized for performance, memory utilization, size or similar constraint. Here we claim that if that part of the code is so important, a complete state transition diagram should be created on that critical part of the code. This code (and its dependencies) should be a subject to a more formal inspection. In addition, better failure scenarios should be discussed to try and create the code section as fault-free as possible, and here 100% feasible branch coverage is the goal, but we suggest to look at other coverage measurements that are applicable (e.g. Linear Code Sequence and jump, that is often called "loop coverage"). We assume that by selecting a part of the code for rank four means that the appropriate additional improvement is conducted e.g. analyzing messaging sequences. Optimized code is often more difficult to maintain, which implies a better documentation is needed.

3.2.5 Fifth Rank – Safety Critical Quality

We are aware that for safety critical code, a number of standards exist that are mandatory and that provides useful guidance for developers. We are just making it clear that this is also the high-end of the quality scale, and gives a perspective for quality. The requirements are to perform formal inspections of all code, perform a FMEA-analysis [151][149], but also to use at least two different static analysis tools and two different memory tools (since they could potentially find different problems).

Profiling tools should be used if applicable, and code should be used with all strict compiler flags set. We have also noticed that executing the code by different compilers can weed out some intricate compiler faults. If the code is safety critical, the documentation must be complete, and include training material. The coverage requirements are at least 100% state transition (n-2) coverage and 100% MCDC [30] coverage. In addition we suggest applying the domain or specific standards, e.g. DO-178B [54] and IEC 61508 [109].

Unfortunately, we have not had the opportunity to explore the rank 5 improvement within Ericsson yet, but some of our external users (medical and defence) explained that the SQR scheme has been valuable for the non-critical code, to make a more controlled distinction of the quality levels of the code.

3.3 The Case Study

Our Case Study was within one worldwide project on a product with distributed design teams consisting of 8 sites/organization and 22 design teams. These design teams were more or less in parallel, and all organizations had different history, motivation and attitude to this quality improvement. We are trying to describe these teams and their experiences in a fashion that could be useful for others with the aim to deploy SQR. We realize that also culture has a large role, where e.g. Swedish developers need to be convinced on a more personal level than more eastern cultures. We have though concluded that developers favor the scheme when they understand the time-negotiating principle of quality, and that the aim is really to make the work developers spend on quality enhancement more explicit for management.

There is a strong tendency that the word of mouth – success of others, is the best motivator. We initially spent more time with people who were willing to use the scheme – and were more “quality-aware” from the start, which made it easier to sell the concept to others if some had success and approved it. Therefore, our target persons were the senior developers in the teams, who would probably do most of the suggested work anyhow, and the effort would not seem so insurmountable. The senior designers are informal leaders, and they

took the initiative to put tools and environment in place for the rest of the team.

3.3.1 Organization A

This organization did only perform SQR, and kept the process as it is. Therefore, this organization is the one of the few that had a quality improvement based only on SQR. This organization had two design teams. The first was known to have better quality from the start than anyone else, and could be viewed as quality aware. They particularly appreciated to move from only functional testing approach to a more structural approach, and were welcoming tools, guidance of what input to select, TDTs, and how to best utilize code coverage. This team quickly selected one person to do the main coverage improvement, but many of designers improved their code according to the concept anyhow. Rank 1 was not performed, so scope was the entire software. This resulted in a doubling of resources, where almost all parts targeted reached rank 3. Here reviews were already a part of the work and test automation mandatory. This resulted in a flawless code, and very few faults were found during the next two test phases.

The second design team had responsibility for new hardware, and most of the personnel were new to design this type of code. Also, some of the code was outsourced, which added more risk. This team was humble enough to ask for a lot of help during the process, which we believe is one of the contributing factors. They were also open for external assessments, which had a positive effect on quality – If you know someone is going to review your results, you put more effort in. At the end, they had trouble achieving the coverage measurements, mostly because of tools problems, and the lack of suitability to do coverage on kernel registers with available tools, but a targeted quality effort brought the measurements up to sufficient levels before release. In particular, we assessed that the conscious review and test targeting were the most beneficial parts of this team's success, since we believe review helped the new team to understand the context of the product. The conclusion was that they reached rank 2 for 60% of target and rank 3 for 10 %. The rest not fulfilled rank 2 in especially for the coverage part, but in most other aspects. This team saved five weeks out of the normal six that was the previous average for this hardware test, for the next phase, functional test, which made these

testers to be moved to other teams, since quality criteria was already fulfilled. The remaining work was with the outsourced part that had not fulfilled the quality requirement and had difficulties in testing their own software in their environment.

The initial cost was expensive in this team (more than double the cost in time of design, but the savings of these two teams were so obvious, in all later phases, that this impacted the entire project.

3.3.2 Organization B

This organization was early adopters of both the new process and the SQR concept and consisted of 4 design teams. They were early selecting strong champions to create an adopted checklist, that later became standard within the entire project, and introduced tools, such as Test Real-time (from IBM/Rational). Much of the test scripts were already written in an internal tool based on tcl, and these scripts were possible to measure using the Test Real-time tools. These teams did an initial assessment to understand their current status and attitude and SQR started a strong internal debate on quality. They took on a too big scope, mandating all new and changed code should reach rank 2, which was followed more or less enthusiastically. This made the internal assessments and follow-up a bit too loose, and the request to provide logs on actual coverage came at a late stage. Three of the four teams focused on increasing the number of tests. Static analysis was for one of these teams considered a good contributing factor spotting 18 real faults in the first run. Reviews were made by all the teams, but again coverage was late to be used as a tool and the test somewhat different for all but one team. This was the team that had the test tool champion, that delivered full automation and good coverage, with many new tests added, but a long time was spent on discussion on how coverage was actually measured. Teams with strong champions had better results. The team with the poorest initial result was the team that delivered code generated from RoseRT. It was a problem how coverage should be judged, since a lot of generated code is unreachable. This team did then really make an effort, and improved their results substantially. Within this organization there were only one team that had a low and a high demand for quality, and these were late adopters within the team. Here a result was also that this team transferred its code to another organization.

The conclusive results for these teams were that about 50% reached approximate 90% of the rank 2 requirements and 40 % reach approximately 70% of Rank 2 requirements. Only 10% achieved rank 2. We believe reason was the wide scope was taken, and rank 1 selection was not targeted enough. Nevertheless, this was considered a substantial improvement, with many strong champions still working within the teams. The main problem we observed was that during the next project the quality approach was lost. We believe that the management did the wrong judgment that *"now when the code has reached its quality - we can cut development efforts again*." This was a mistake, that even if all touched code was a target for quality improvement, it was not completely fulfilled, and never reached rank 3, and also a new project will target completely different parts of the code, that modifications and the added new code will still need its targeted effort. We also discovered the fact that all new code must have Rank 2 which we believe is an impossible task with the time given and will work as a discouraging factor.

3.3.3 Organization C

This organization consists of 10 different design teams, whereof 8 of 10 did attempt SQR and two teams "cheated" by filling in fictitious values. This became revealed when we reviewed failure reports, where all teams have improved substantially except these two. We guess that no one would believe that someone seriously would cheat, and rather bought the talk of "these components being so special". This organization had weak initial interest, and made a very minimalist checklist, which was later abandoned for organization B's checklist. The first two teams got no extra time, and not until organization A and B had started to show good results, this organization took a serious look. In addition, the initial champion had moved away to a new role, and the managers were supposed to drive the improvement failed in all aspects. The top project management had to assure that time was really given to achieve the requirements of SQR 2, and the contract principle of the component test plan won developers. Then the team started to catch on, by creating test environments, using test tools, and targeting the right components. Here the "second best" persons were getting real results and could actually see how they were saving time for themselves, which finally

convinced many people to change. This team did also have a lot of rank 4 components, but the lack of sufficient tools, substantial stubbing, etc impacted many teams to get the real success. It is still hard to judge the result in factual numbers on coverage in these teams, since they were sensitive to outside assessment, but the test maturity has improved tremendously for this organization – which is the most important result. The quality for 8 of 10 teams got lowered to 10% of their earlier average and most of the problems were now found within the organization, instead by the next phases. An additional observation was that they also introduced several steps of testing within the development phase. Even if the new process is used, they moved during this project from one test level to wanting four internal test phases before release outside their organization. The four levels are designer (stubbed) software, component, multi-component and functional test. The conclusion is impressive, and the remaining faults are often so intricate that they are hard to trace and debug.

3.3.4 Organization D

These were the earliest adopters of the concept, being suppliers in an external organization they saw this as a requirement, and thus created a checklist for easy self-assessment. In one sense, these teams were the most experimental to the concept. In practice, we assessed that they did well in all aspects of SQR that they personally believed. They never understood contracting principle internally in the component test plan, but we believe they targeted software, even if only a limited extra time was added. The main reason is that the benefit of improved quality would result in less work for the design team, since the savings would be at test levels at later stages and outside the development organization. At this time, that was not a positive factor, yet we could see a will to make this happen. The review concept, as a quality contribution, was never taken seriously, and treated as a hand over between two developers signing off the code. In all other aspects, we believe the SQR was followed. What was impressive in this team was the management engagement, and that aim to perform well, where some energy was set on test. They claimed themselves having great success with the scheme, but it has been hard to review from the outside. The result of this code was in large parts outsourced further, due to economic pressures. The most

interesting result in addition to the quality improvement (where all selected targets did reach rank 2 according to them), is that the remaining 75% of the faults were related to memory problems for this area. This was despite the fact that memory tools were used at the lowest level, an interesting indication that memory problems can remain (and additionally must be revealed) in later stages of testing.

3.3.5 Organization E

This team was a very tight team with partly "unreasonable" quality demands (all or nothing) approach to their software. They adopted the new process and SQR at the same time, and what was particularly interesting is their approach to static analysis were one approach that found many faults. The team dedicated two days cleaning all static analysis warnings and had one champion that had learned the tool to support the team. Coverage figures were debated since they wanted the figures to work on all components. Since they failed to target their code selection enough, the approach fell more on individual designers' time and interest to improve their components. In many aspects, they did not fulfill rank 2 at all. In later analysis it became clear that their budget was structured on maintenance, and a too good quality would have lowered their budget (and giving them less work), which is not a real internal incentive of becoming too good for the organization. This indicates valuable lessons for contracting of work and quality incentives of software.

3.3.6 Organization F

Organization F consisted of two teams. This organization could easily recruit people, and many designers and testers were relatively new working in Ericsson. Good management and early teaching on testing made this team very interested in this quality improvement, and they were very happy that there was an increasing quality demand on the product, as they saw as an opportunity for them. These teams embraced the concept of SQR and were listening carefully to any testing advice. The most problematic issues were the component test plan and the contracting principle, which they felt new in their role and it was difficult to make a case with their management. Yet their

focus and interest motivated them to have good test behavior early, and extra persons were added to the team instead of giving them more time. Automating test was a conscious decision of the entire team, which also proved valuable. The result showed that rank 2 was achieved for most components and the quality was substantially improved. Since there were some hesitations with the accuracy of this results of which we could not assess, we are hesitant to dwell on measurements for these teams. Also, no later follow-up has been conducted, but we do know for a fact that SQR concept spread further within their organization.

3.3.7 Organization G

Organization G was a mid-size team and has no introduction or training in SQR ideas, except what was written. They did instead make their own "unauthorized" checklist that was a simplification. In later review of the checklist, we found many principles that were interpreted in a questionable matter. This team had not understood the word "feasible" coverage, and delivered 100% statement coverage of all their components. Yet, no other functional testing was made, which resulted in fault-prone software that had many integration problems. The remedy was sending several seniors on place to try and educate the team. There were no particular SQR assessment or follow-up in this team, but we mention it to point as an example of how easy it is to misunderstand and misuse a good concept, achieving "on the paper" some metrics, and yet failing the quality.

3.3.8 Organization H

This final small team of 6 persons and one tester, but medium size software, was an enigma. In the initial teaching of SQR, we believed they had not grasped it, but on follow-up, they exceeded all our expectations. In later reflection this should have been judged as winners, since they at their first meeting could present factual information on their actual quality, which is a good indication of control. They grasped the internal (somewhat secret) ambition of the entire SQR project that at least half of the number of current faults should disappear after this improvement. The thinking was if all

individual designers improved, the overall sum of faults would be substantially less. This team had only 56 failures on their software (and only acknowledged 26 of them as true software faults). The goal was set at 14. There is not much to say except that they followed SQR with their own checklist in all aspects, and the final result was that only 6 faults were found in later stages, but also this team managed a much earlier delivery and the code was considered as a high quality code. The team was definitely a very quality aware team, where all components selected reached rank 3, and 2 according to plan and it was hard to find anything that could have been done better given the limited time and resource.

3.4 Conclusions

In conclusion the failures in system test dropped to 10% of the earlier versions, and an independent measurement showed that the SQR concept saved 67% technical hours, diminishing the time for maintenance substantially. In practice, a reduced failure administration from hundreds of problems a week to 2-3 failures every second week proved to be the factual result.

Most of the savings came in all later phases of the project, which were different levels of testing (functional and system testing), where a diminished test time took place, and weeks were saved in many of the test organisations receiving the software. If the quality is good (great) from design, our assumption is that all sub-sequential test levels, including corrections of code, administration of failures etc will save a lot of time. This was obvious in this project during integration. The reason is that a quality product will be faster and easier to install and integrate, and test suites will have a faster throughput, since less failures are found, and thus less regression-cycles and re-deliveries have to be made, all factors that save time.

The quality improvement during this 18 months project actually challenged the system testers to re-design their test cases, and left room to handle a lot of change requests, get control of back-logs and basically return to a more reasonable working situation. It is no secret this product was pushed a bit too hard in the time to market race for release, and a bit too many quality problems were a result of cutting too many corners in earlier releases. Now this product is viewed as

the best in class, when it comes to quality compared to its competitors. We definitely think that the new process and SQR together have unquestionably moved fault finding to the earlier phases in development life-cycle. It is easy to view the result, where we could see that now the organisations (A-H) finds their own failures to 85% and the next (internal) customer and external customer only finds 15% of the failures. The figures before this improvements were vice versa.

An interesting finding is that additionally 2 design teams did claim to use the SQR method, but had falsified their coverage results in the check-lists given. They were therefore excluded initially from the study above. These two units did not change their behaviour at all, but changed their process. Since these two teams also stood out in their lack of quality improvement (they remained at the same level as before), but they did change both their integration method and project way of working (process changes) – other conflicting indicators that could be a part of the quality improvement explanation. Because they remained at the same level – and they did NOT use the SQR method this could be argued as an indication that it really was the SQR method that contributed to the quality improvement.

We are still investigating the possibility of publishing an aggregated version of the coverage results in relation to the fault-failure data. This would provide further evidence of the findings and the quality relation between coverage and failures. By no means, are the SQR combined basic methods of using static analysis, reviews and functional test in combination with coverage are sufficiently investigated.

A particular motivational drive was that it seemed that most design teams improved their quality by increasing their statement code coverage, but there was also one design team performing 100% statement code coverage, that still had very poor quality. This was a puzzling result. A similar variation of data could be found on how much improvement a static analysis tool had on the quality. Since this was not systematically measured, it is hard to draw any conclusions, but to more a series of actions as “indirect” indicators of quality. If a design team did perform static analysis and reviews regularly and consistently worked with code coverage improvements, the quality “awareness” and focus seemed to remain higher than in teams that did

not. It is of course hard to compare the teams, since code competence, system competence, background and their personal motivation and interest in showing good results are some of the background factors that are outside this study result.

3.5 Discussion

The most common question we got during the introduction of this improvement *is: What is a component?* Our explanations have changed, where we have ranged from: managed item in the Configuration Management tool, a conceptual item, the smallest executable (identifiable) piece of code, to a more general item with a clear interface that could be executed in isolation. We view the debate as a decoy, and this will be settled when starting to work with the tools. If the component is set on a too high level, the coverage result is too difficult to achieve.

The second most common question is *what coverage is enough*? Again, all code is not equal, which is an important factor. 100% feasible coverage (assumed statement) does not mean tested. This is clearly shown when discussing conditions, loops, and input, and developers become aware of this fact. Therefore, we allow differences of coverage within the component. The follow up question is *what coverage should be demanded?* Our approach to feasible testing is strong, but the question reveals the wrong attitude. It should instead be: *How do you know if you tested enough?* Stopping criteria is a known and interesting discussion topic, but in fact, there most common answer in industry is – when time run out, instead of fulfilling tasks to completion. This is because development time is often underestimated, as well as the time needed to test your solution, debug and correct the found problems and retest them again.

We also believe that there exists no real answer on how effective reviews are as a generic method. It is difficult in practice and depends on how you perform the reviews. Furthermore there it seems to be a practice used in high quality software teams, and lacking in low quality software teams. Indirectly it indicates that people are taking the time to learn and reflect on the code quality developed, which is the result of responsible behavior.

Static Analysis tools can be debated [1], but we review the use of this in the same manner as reviews, it is more common with quality producing developers.

When assessing our results, and especially looking at other projects attempting to reuse the concept of SQR in their organization, we cannot hesitate to ask ourselves: "*Why all developers do not have same success in quality when adopting SQR?*" and we can take this further: *Why are organizations failing to get such good results as our project?* We realize that the SQR measurements many collect are not comparative, since they were not preceded by an assessment that showed component test is a problem. Secondly, we believe that often the initial Rank 1, making a targeted selection was ignored, and a too wide scope was selected. Scope being too large, ignoring the contracting principle – meaning, given sufficient extra time to work on quality aspects, and a lack of motivation and knowledge (in tools used, methods used and testing in general), seems to be the main reasons to fail with the SQR scheme. If the scheme became "just another checklist to fill in" before release, it possess very little practical improvement meaning for the developers. This might also be a cultural view of the developers, whereas during stronger measurements of being on time than having the correct quality is in use, the quality will always take second place.

Replication of the method – without the influence of the researcher has resulted in a diversification and change of the initial approach – and also produced a variety of results. The main problem of replication is the relative lack of detail in all descriptions used, and the definitions are not on right detailed level, allowing somewhat liberate interpretations. The most common change has been adapting the checklist, and made a goal "that all software should fill in the status". This means that often developer has no motivation to spend more effort on particular parts of software, and there is no deliberate selection of targets to improve. This causes a generic down-grading of the scheme. Secondly, the underlying ranking has not taken place, and adequate extra time to do the improvements has not been given. This lack of focus impacts the actual possibility to perform an improved testing. In fact, minor improvements have taken place when the check-list is not used as a deliberate planning and targeting of improvement, but has instead becoming a general documentation, which instead are actually adding burden, without much special

changes in the ways of working. The most common change is that static analysis tools are used more often, and that statement coverage might be measured more often. In fact, the acceptance of low coverage seems to be a commonplace attitude, since the assumption that "statement coverage is anyhow not a good indication of quality" is prevailing. Arguing that releasing code that has never been executed is a serious danger – since it could be filled with problems seems to have some impact.

We can also see attempts to make SQR into a quality measurement tool – given each rank and item fulfillment a value, and aggregating the result as a measurement for the component. This has locally been showing some improvement, but does often miss the actual target – quality software – instead of fulfilling measurements as a goal of the work.

3.6 Lessons Learned

Ericsson did not only cut delivery time to less than half, but the quality improvement was substantial. We believe strongly that stepwise improvement is the best approach, but there should not be too many steps to achieve, as this adds complexity. We also conclude that component test is probably a focus for many organisations with time to market software, in addition to system testing. We believe the focus is the developers that at least within Ericsson have a major quality impact. Therefore it is crucial to provide developers with better tools and test environment.

Some lessons learned from this project:

- A code with 100% statement coverage could have worse failures than one with only 70% statement coverage.
- If statement code coverage was combined with TDTs, the results seem to be better.

Chapter 4. The Test Design Technique Comparison Framework

4.1 Summary

In this chapter an experimental framework for comparison of TDTs with respect to efficiency, effectiveness and applicability is proposed. Some of the problems of evaluating or comparing TDTs in an objective manner are highlighted. The basis of comparison frameworks is neither a well described process in research nor in industry.

The planned process for this multi-phase experimental study is described. This includes presentation of some of the important measurements to be collected with the dual goals of analyzing the properties of the test technique, as well as ways of validating our experimental framework. The experimental framework aims to better make a distinction between how the experiment is set up and the techniques are compared, than earlier work has done. By carefully define how to compare TDTs, and then supply a detailed process of how to deploy this framework; the results of the evaluation would be generic, and thus more useful. This means important measurements for analysis are prepared, as well as a proposed way to validate the framework and thus constantly improve the juxtaposing. Most current research compares their technique to either a “random” result or one other TDT, for a specific and limited code sample; containing few faults, where the technique investigated usually have strengths. In fact, how to set up a comparable framework and what to measure is a way to define how we would achieve comparable results for our research.

4.1.1 Context of this Study

This framework has formed a basis of approach for our TDT investigation, as the result was partly supported by a presentation at ICSE 2006 in Shanghai by Basili and Elbaum [13]. Our main point is that the software under test has a great impact on the result, must be taken into account. We were still a bit naïve in the undertaking and consequences our proposal had. Something that will be further explained and discussed in the consequently case studies (Chapter 6). Our intention was initially that as a byproduct that our framework should be able to identify and minimize heuristically aspects of applying a TDT, by the means of automation. This is by no means straight forward, and our initial attempt on describing the TDT in its components and relating that transformation into automation is just at its infancy described in the TDT taxonomy (see Chapter 13).

Because some fundamental problems in deploying the steps of this framework, we must conclude that as important as we feel the intention is, it is fundamentally difficult to fulfill it, in all its aspects. In particular the fundament of basing the evaluations on a series and variety of injectable fault types, and understanding their propagation in a variety of systems, means, we have also failed in the ever so important groundwork for a fair and generic comparison of the technique, and must subsequently also join the class of limited results. In essence, the framework will be described as a theoretical goal based on our initial approach, thus describe the entire process, steps to take, and what to measure in a straight forward manner. It did at the time define how we were planning to proceed, to achieve our overall goal. We can also see how that we continued with the first sub-process thus the next step in study 3 highlighted in chapter 6, trying to utilize this process and particularly phase 1.1 and 1.2. At study 3, we realized the limitations and problems of the existing classifications of fault and failures, not available in a structured way to be used in this manner. This will be discussed in further work.

4.1.2 Design of Study - Research Method

This work is based on investigation of a series of TDTs research papers, identifying the fallacies with the current methods, and attempting a new more comparative method, that would yield a more generic result. This is purely a theoretical construction, and provides no evaluation in itself, but has shown providing a clarification on how and what to measure when evaluating TDTs. It describes a multi-phase experimental study, where we show several areas necessary to deploy before a fair evaluation can happen, such as having a clear fault classification, and providing a series of software code of different origins to make the results valuable.

4.1.3 Validity Threats

Validity of this proposed framework is only done manually, and only at parts, where the automation of the TDTs are left for further work. Any other method validating this framework other than using it, and then empirically estimate if the result seems feasible is farfetched. One could probably provide a critical assessment of it evolution methods used. We suggest that using the framework would provide a better know-how of its strength and weakness of the framework. If the result was generic applicable, the resulting proposed guidelines could be used for all types of software, regardless of domain, language, purpose and development method, organisation and deployment skill. There are few theories that stand up to such a scrutiny over time. Yet, the contribution here highlights the fact that even though we spent decades using and evaluating TDTs, it is by no means a mature science, and TDTs has not been sufficiently evaluated enough in an industrial context, where number of faults, and the size of system under test are much more complex and often larger, but at the same time, much less controllable than an experimental research set-up.

We noted that the applicability was especially important, since we knew that techniques can be applied subjectively. We had hoped to define the techniques and their variants as clear as possible, so they give measurable and objective results. This would also clarify what techniques, or what circumstances, are not measurable. Even if substantial time was spent on this issue, we were still not succeeding in full to use and deploy the techniques similar throughout the studies.

At least within each study, one can assume the techniques were similar, since they were performed by the same persons. In study 8 and 10 we could definitely see the benefits of using the same persons doing the evaluations. The validity then becomes a systematic fault. We have used several ways to improve the validity of the result. We used studies in a more laboratory setting, and treated the results through statistical analysis (empirical evaluation) of our measurements. The evaluation of the framework is through its deployment, where it was performed partly, in studies (see study 4 to 10 in Chapter 6 to 12). These studies performed as the basis for our guidelines and TDT taxonomy. We noted that our evaluation is purely manual and all attempts to automate the TDTs have been left for future work. Our aim was to further improve the validity of the result, by repeating the experiment with both controlled code samples and uncontrolled samples; unfortunately, we never pursued a fully controlled code sample. The technique was aimed to be repeated for untested code samples with different fault types injected. This attempt was particularly used in study 8 and 10. Finally, we aimed to manipulate the information on the number of faults injected (that could be expected to be found). The goal with this manipulation of information is to avoid what is called “self-fulfilling prophecies”, i.e. if you know there are 5 seeded faults, you are looking for 5 (even if 10 were seeded!). We failed to conclude this approach.

4.1.4 Contribution

The main contribution is providing a framework of comparison of TDTs, and presenting measurements to be collected whilst analyzing the properties of the TDTs as a dual goal. The study highlights some of the problems of evaluating TDTs in an objective manner. It provides a process for how to proceed in a stepwise manner, and justifies the approach.

4.2 Introduction to the Test Framework

We set out to find a way to compare TDTs in the aim to answer the two questions:

- *What type* and *how many occurrences* of failures does a *particular test technique find?*
- *How effectively* and *efficiently* does a particular test technique find these failures?

We are interested in understanding the *applicability* of each technique, since there are many variants of how techniques are utilized at different levels in the test phases, and how the technique can be applied determines the outcome. Are some techniques more efficient than others are? Are these techniques better than the "heuristic" TDTs?

We differentiate between faults and failures, where a fault is the manifestation in the code, and the failure what becomes visible during execution (testing). We are interested in all faults, since they might get exposed and propagate to failures. Faults are often hidden, but are important since they tend to propagate when the code is reused in another context or when the code is executed in a new environment. One aim of our research is to provide support for selecting which techniques to use, when time and resources limit us. These guidelines will complement and improve the current test case design in the industry.

In this chapter, we present a framework for comparison of different TDTs. We also propose a process for how to perform the evaluation. Our process is straight-forward:

1. Prepare code samples with known faults.
2. Select a test technique.
3. Perform the experiment, by applying the technique, and collect data.
4. Analyze data, compare TDTs and evaluate result of collected data, and repeat experiment with new code samples (new faults injected), until achieving sufficient amount of significant data allowing possible conclusions.

With these data, we can analyze and compare the efficiency and effectiveness of TDTs and also obtains a basis for evaluation of the applicability of the techniques. The conclusions about applicability

are based on evaluating the TDTs both manually and automatically. As a byproduct, our framework could also identify and minimize heuristic aspects of applying a TDT, by the means of automation, and also aid in evaluation of automation.

4.2.1 Related Work for the Test Technique Comparisons

A recent overview can be found in Hyunsook et al [105], discussing some important infrastructure of test experiments. Basili and Selby [13] have performed an evaluation of software TDTs where they compared code inspection with functional and structural testing. Kamsties and Lott [131] and Wood et al. [216] have replicated this work. They found effectiveness-related differences between functional and structural techniques depending on the program to which they were applied. Hetzel [98] and Myers [162] did similar comparisons to Basili and Selby and with contradictory results. Hetzel favored functional testing and Myers did not. A shortcoming of these three experiments is *how* test cases are selected (the definition of e.g. functional testing is not enough if only described as "based on specification"), which is too coarse. Their result differs, and it is not surprising that they arrived at different conclusions, since TDTs can be applied differently (by different people), and the programs were not exactly the same. Basili and Selby's work has been replicated [216] showing these weaknesses. Rather than evaluating two techniques against each other, it is for us more important to identify the usage of a technique; for what faults (failures), what level and what systems they are applicable. We agree with Juristo et al [128] that most studies of TDTs are not sufficient, for instance lacking in sample size and number of samples. Our work focuses more on fault types and how they propagate to failures, and as a consequence, we construct our framework differently.

There is an abundance of TDTs. Rothermel and Harrold [185], from whom we reuse and adapt some concepts, inspire us. They are using a comparison framework for regression TDTs, which can be viewed as a test case selection method. However, regression testing represents a subset of techniques that is based on knowing the expected test result.

In the more general setting, we consider test results are typically not *a priori* known.

Three different approaches for comparing and evaluating TDTs are commonly used:

1. The use of coverage (often suited or invented for a particular technique) is the most common [79][220].
2. Comparing one technique with another in the same group, also called intra-family comparisons. This is often used when a new technique is created as an improvement or evolution of the original technique, found in [79][128][209].
3. Comparing a particular technique with others out of the family of techniques, also called inter-family techniques [13] [128][131] [162][216].

For practical reasons most TDTs have been evaluated using small code examples, but there are a few examples of experiments on larger samples.
We believe that any test technique finds failures and reveals faults. The difficulty in evaluating TDTs lies in the estimation of how effectively they do that, and to what extent the technique is scalable and possible to generalize to all types of codes and domains. We must assume that fault and failure types and their distributions vary a great deal between different software systems.

TDTs can be classified or grouped in different ways; e.g. into structural, functional and non-functional TDTs *or* into black-box, clear-box and white-box techniques. These groups are however not sufficient for comparing TDTs. We will explore them further. For example, one group is characterized by usage of input data as the common denominator, where random, fast anti-random, and data selection techniques are explored. Another group of techniques focuses on evaluating a complete set of test cases to cover all possible paths in the code, for example mutation TDTs [50][169] or genetic test algorithms [126] [206]. One can also find groupings based on how to test. For example, automation is a separate group, where TDTs that can be automated are in focus. Code inspection and static analysis tools are other examples.

Recently, Vegas and Basili [201] presented a characterization schema for the selection of TDTs. The main focus was to build a repository containing all known and relevant information related to various

TDTs, along with multiple views of test technique designers, testers as well as researchers. This is kind of a complementary issue to what we are interested in and could definitely be beneficial in selecting the subset of TDTs to start with for further detailed evaluation in our framework.

Juristo, Moreno and Vegas [128] have done a comprehensive review over 25 years of software testing techniques. We share their conclusions that the area of testing technique comparisons lacks maturity and that many of the results are not scalable, statistical sound, repeatable or just not detailed enough.

4.3 Overview of Our Proposed Process for Test Design Technique Comparison

4.3.1 Research Questions

Our main research question is if it is possible to compare different software TDTs in general, by using the proposed evaluation framework, and its associated process, described in *Figure 1* below.[2] To be able to evaluate the different TDTs we need to tackle a variety of problems. We need to create a controlled experiment that will be large enough, and sufficient for statistical purposes. This is the first phase of our process, preparing code samples to be used in the experiment. All of them are prepared with sufficient number and type of faults that will propagate to failures. This phase of the process will be discussed in more detail in Section 4.4.2.

The goal of the second phase in *Figure 4.1* is to select a test technique and, in phase 3, to apply it to the prepared code. Here the actual experiment starts, and so does our measurements. Our purpose is to investigate different variants of a particular group of TDTs.

[2] We feel that not enough attention has been given to objective comparison of the core of the test design. The evaluation process is not straight forward as proposed. In [203] different comparisons frameworks have been investigated.

The aim in the second phase is to be able to select a test technique, classify the technique into a group or place an invented technique into a group. We will explore the published variants of the technique, and the different ways of how to automate the techniques, which in itself will be a challenging task. During the measurement phase, 3, we will use different techniques to obtain a scientific evaluation, using quantitative, qualitative and empirical research methods (see Section 4.4.5 for further details).

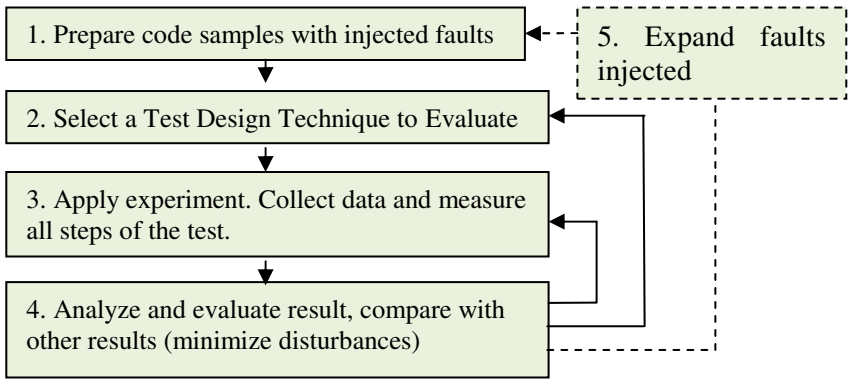


Figure 4.1 Overall process of evaluating TDTs

We will thereafter evaluate the result in phase 5. After the first application of this process, we expect that we may need to adjust some details. Then our aim is to perform this process over and over for several TDTs. This will start a long endeavor, and we hope to provide the community with more and more scientifically based results on the merits and weaknesses of existing and emerging TDTs.

4.3.2 Prepare Code samples

The idea of the first phase (see *Figure 4.2*) is that we need to create a sample code that is large enough (contains multiple components), and is representative of a real system at this level. Our approach is to select real product software, and re-inject faults of different fault types (from production) into the code, to be able to check, if and when the fault propagates to a visible failure (under what circumstances). The aim is to be able to check if there is a test technique that can find this particular failure. We need to be able to seed a number of

different types of faults that represent a number of different types of failures.

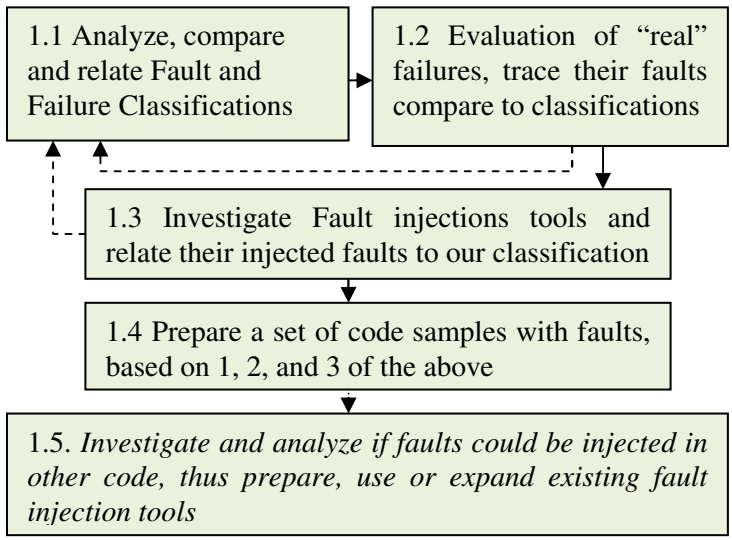


Figure 4.2 First phase: Preparing experiment code

To do this, we need to investigate different fault and failure classifications (*Figure 4.2, phase 1.1*), and the relations between faults and failures. Secondly, (*Figure 4.2, phase 1.2*) we need to compare our own reported failures, trace their faults, and validate or improve the classifications. Thirdly, (*Figure 4.2, phase 1.3*) we will investigate different fault injection tools to understand what faults they inject, and which of these will propagate into failures. This phase might also find new faults and failures to validate and check the classification. We will use fault injection tools to aid our evaluations of TDTs. Fault injection is used to either test the system's ability to handle faulty components or software, or to check if a test suite is complete. To our knowledge, fault injection techniques have not been used before to aid in evaluating a test technique.

We realize that we probably will need a variety of copies (*Figure 4.2, phase 1.4*) of the software with different sets of faults, to minimize internal interference of the faults. How this will be done in practice must be further investigated. Then we have an option to investigate the fault injection tools further, to see if we can expand them, adapt or

possible find uses for them in our aim to verify system architecture features (*Figure 4.2, phase 1.5*). This is not necessary for the purpose of this investigation, but might be a bonus, which is the reason why it is being written in italics in the figure.

4.3.3 Industrial Challenges for Faults

Fault seeding and fault injection are not trivial. There is no straightforward classification that helps you to decide what faults to inject in an experiment on evaluation TDTs. One can get inspiration from Mutation testing [169], and the Mothra [48] defined faults. The area is research intense. A particular fault class is investigated along with its corresponding remedies. At conferences whole sessions and tracks are devoted to the subject (e.g. at ISSTA 2004 and ICSE 2006). To investigate this further, it is common to define a limited set of faults that serve the particular purpose of investigation. This area of research is also interesting for compiler-assisted techniques, static analysis and dynamic analysis. It tries to find answers to the following questions:

- What are true mutants of faults?
- What are the semantics of faults? [168]
- What impact does a fault have?

Yet the industrial questions when analyzing these faults (often mixed with failures, and other causes) are still not answered satisfactorily. For practical purposes, simple classifications are often in use e.g. ODC [35], Beizer's Bug Taxonomy [18], or IEEE 1044-1993 Standard Classification for Software Anomalies [111].

The problems with these classifications are in terminology, scope, frequency and impact on the system (severity and semantic evaluation). The challenge is to answer which are trivial and which are non-trivial faults, and what impact a particular fault has (what causes failures). Can the fault be generalized to be used everywhere (or how bound to the domain/semantics is the fault)? What is a good and true fault classification sufficient to juxtapose any two techniques? The answers to these questions are still left unanswered.

In the systems we consider, we can currently identify between 1 to 64 files that is involved to correct a failure. Is it the "same" fault in all 64 files for one failure? Again, it is difficult to answer, and hints at a

more complex fault seeding and fault injections must be considered. It is not clear what constitutes sufficient information for a designer to be able to classify a fault (from analyzing a failure) as of today. What is needed is to aid understanding, prevention, and support root cause evaluations and debugging. We know little about scalability and impact of faults and we need to better understand how do faults propagate to failures, and what are appropriate levels of testing to localize them.

4.4 Selecting the TDT

After we have prepared our software for our experiments, we can start investigating the software TDTs (*Figure 4.3, second phase, 2*). We will then explore how to select a test technique. One group of TDTs could be those using static analysis tools. These tools often have a set of different options. In our experiment, static analysis is not our main interest, since we focus on dynamic testing of failures and not faults. We assume that all faults injected are faults that will *not* be captured by a normal compiler, and our preparation of the fault injections is adapted accordingly.

The first phase in *Figure 4.3*, phase 2.1 might seem redundant, but should not be omitted. We assume that a general introduction about what testing is, test cases, test specifications, designing test cases, test level etc. already exists. We suspect that each TDT is not uniformly defined and applied, but could have variants and could also be applied differently in different context and at different levels.

It is important that we clarify this, and also that we choose what variant and at which level we are going to measure. It will probably be easy to compare more than one variant of a technique, and compare the same technique at different levels. What impact this has on the result, will be evaluated and will also be measured to explain the applicability of the test technique.

There is often a specific interest or purpose of evaluating a particular test technique, based on the assumption that the technique will be more effective. It is possible to shortcut this process and go directly to phase 2.3 (*Figure 4.3*). Regardless of our technique, it could be wise to try to understand what types of failures and faults a particular

technique can be expected to find. We assume we have a mechanism of placing the technique in a group, and that these groups can be redefined if we discover that this is necessary. Groupings of TDTs exist in many forms, where some examples are in [19][128][202].

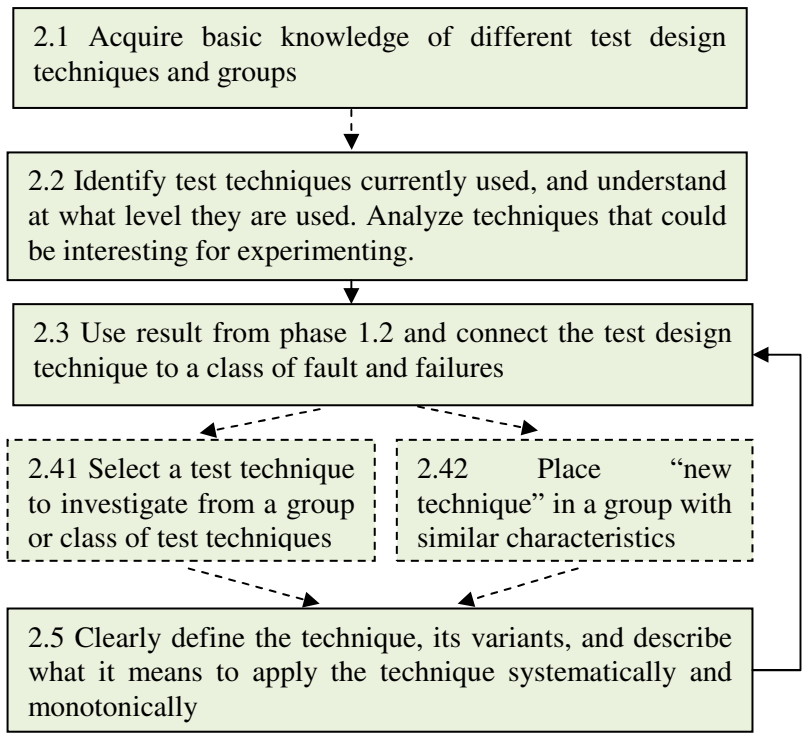


Figure 4.3 The second phase: Process of selecting a TDT to evaluate

Then we have two alternative approaches. First in phase 2.42, for a new invention we need to get comparison material and place the technique in a group, or using the simpler approach is to select several techniques to compare from a group (phase 2.41). The idea is that it will probably be difficult to compare techniques between groups, since they find different faults. The final step in this phase might come as a result of 2.1 and 2.2 but nevertheless in phase 2.5 we will understand the applicability of the test technique. This may only be useful during certain circumstances, but the idea is to describe what it means to apply the technique systematically and monotonically.

Systematically means that the technique can be applied similarly for all cases, and monotonically means that, if it is applied, it will give the same result every time. The problem here will be to identify what it means for a test to be successful, i.e. to determine if the result is correct or not.

4.4.1 Industrial Challenge of Test Selection

As stated in [128], the TDTs are usually evaluated in small code samples. There is a lack of well-founded guidelines suggesting when different TDTs are effective, efficient and applicable. Our focus is to find a way to traverse the TDTs, and try to establish the maturity of each technique, by reusing results, and extending them into an industrial setting.

We will focus on phase 2.3, where we suspect that TDTs relates to faults and to their propagation as failures. We suggest that this is a new approach. The challenges here will be to answer the question: Can a test technique be assigned to a particular fault or class of faults or does it focus on how the failure is visible? What are the guidelines that make a technique more or less useful in a real system? If we can answer these questions, it would mean that it would be possible to assign a particular technique to a particular phase in the testing, and also, that it would be possible to evaluate a system failure (or fault) and define the "best" test technique. The efficiency of different techniques depends on where and how they are applied.

4.4.2 Applying the TDT

When the TDT is selected, and the experiment sample is prepared, we can start to apply a particular technique (*Figure 4.4*) for the experiment, which will be measured and evaluated during the entire process (*Figure 4.4, phase 3.5*).

We assume that we have rudimentary information about the system and testing in this system (*Figure 4.4, phase 3.1*), and a short introduction about the aim of the experiment – to evaluate TDTs. This phase has no impact on the research, and is therefore in italics. Then the technique must be understood, and both a short introduction and a more in depth search of references in the area is needed to get a

theoretical understanding of the TDT (*Figure 4.4, phase 3.2*). The test technique is applied to the code sample(s) by creating test cases manually using the technique (*Figure 4.4, phase 3.3*).

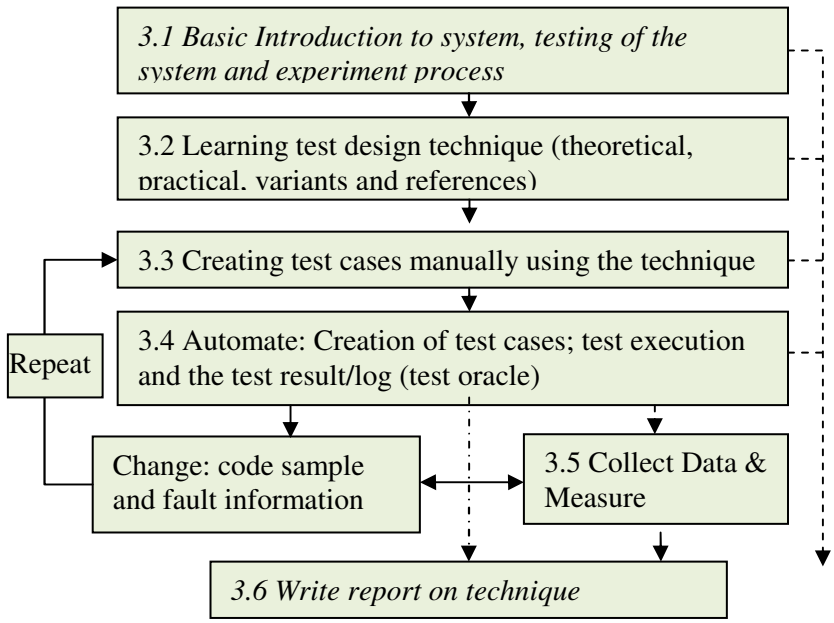


Figure 4.4 Third phase: The experiment and data collection

The work of automation is attempted in phase 3.4, which includes automating the creation (finding) of a test case, automating the execution of the test case, and finally automating the analysis and evaluation of results of the execution. Some techniques may not be possible to automate at any or all of these automation steps. The data collected during the earlier phases, will be processed, analyzed (*Figure 4.4, phase 3.4*) and summarized (*Figure 4.4, phase 3.6*). This report is a part of the experiment, but is not necessary for the outcome, and is therefore shown in italics.

4.4.3 Feasibility to Automate a Test Case, and Industrial Challenges

We have found some guidelines on what areas are feasible to automate and where the main problems are. Yet, this is not a trivial task. The way a technique is automated can determine its real value, and usage. Many research constructed automations are valuable, but moving the result into easily applicable system for testing is a different matter. The challenge is that many tools are too specific, and not generalizable, or are not available for the industrial community. A lot of research is often government funded, but the result and details (often necessary for reproducing the research) are unfortunately hidden. These reasons substantially reduce the value of much of the available research results. We hope that we will be in a position to make details of our results available.

The automation of a test case consists of several steps:

a. Test case selection criteria manual, automated (search, generate test cases).
b. Test case minimization if generated (based on criteria, often coverage) and completeness
c. Test execution (manual, automated)
d. Test case evaluation (test oracle, manual, automated) (often the real hurdle with automation)

The industrial challenges for the steps are:

a. That selection may not work on larger systems. Automatic generation of test cases can be easy, but is often time consuming. Different tactics of limiting the search is often applied.
b. Reduction of the test set is usually no hurdle in itself, since test cases are just discarded if they do not contribute towards the fulfillment of a criterion (e.g. coverage). However, for a complex system, it is difficult to efficiently instrument and measure data-flow coverage and control-flow coverage. For small systems, completeness is achievable, but for large systems, there must be criteria to cut, prune or determine when the suite is sufficient. Many researchers ignore this step.
c. Execution, from a research stand point, poses no trouble, but in reality, the way *how* this execution is automated could make a huge difference for how useful the automation will be in a real

system. Areas that need better support include: Manual intervention (which is often needed for determining test results), logging, tracing, and instrumentation (which are often used as a part of the test case execution), and adaptations and workarounds (to make the test execution possible).

d. The evaluation is problematic since in a large complex system that often use fault tolerant and fail safe, self managing architectures, a fault is easily hidden and might not propagate immediately and visibly. In this area, detailed research on real systems is needed. The fault might be dormant, and then with reuse in a different context, propagates itself and cause failures. Therefore, one is often forced to use manual evaluation, which is not feasible for large sets of test cases, especially – it is contradictory to the aim to get more and better test cases.

4.5 Measurements, Evaluation and Validation

The testers will be evaluated during the process. This is probably a bit subjective, but can aid in interpretation of the result. We aim to reduce the Validity threats of the results, i.e. reactivity, researcher bias, and respondent bias [215]with different types of triangulations. Several tests and evaluations of the system (along with knowledge-based questions) will be performed. Then the work with the TDTs starts, together with the evaluation and data collection. For all steps within phase 3, the following measurements and control questions will be asked. We have grouped them according to what they aim to measure. In estimating applicability, measurements from efficiency and effectiveness will also be used as a complement.

4.5.1 Measuring Efficiency

1. Actual time, i.e. planning, implementation and execution (manual & automation), (calendar-time and estimated time) for each phase (hours/days) (Quantitative)
2. Time to detect faults and/or failures, and also time to identify the fault type (minutes/hours) (Quantitative)

3. How long time it takes to find the first fault or failure in minutes (Quantitative)
4. The subjects own judgment of every task in the process (Easy, difficult, poses secondary problems etc.). The assumption is, what is easy is also fast. (Qualitative)
5. The time to manually create test cases (for one, the first, and many variants) (Quantitative)
6. How many unique test cases, and instances of the test case is created (number per test case/number of variants) (Quantitative)

4.5.2 Measuring Effectiveness

1. Absolute numbers of how many of the seeded (and other) faults were found (isolated) compared to injected faults. (% faults detected, % faults isolated). (Quantitative)
2. For each fault found, identify what type, how many of the faults are isolated, and faults severity. % faults detected/type, % faults isolated/type % (Quantitative) faults of each severity (A, B, C) (Qualitative/Quantitative).
3. Estimation of "coverage" in % and measured where possible, dataflow and control flow coverage using the technique, as a support to the effectiveness of the test case suite. (Quantitative/Qualitative)

4.5.3 Applicability of the Technique

1. In what phase faults are found (distributed) over time (quantitative)
2. The subject's own judgment of every task in the process (Easy, difficult, poses secondary problems etc) (Qualitative)
3. The ease of learning the technique in phase 3.2 (Easy, difficult, poses secondary problems etc) (Qualitative)
4. Levels (code, component, integration, sub-system, system) where the technique is possible to use (Qualitative)
5. The generality of the technique, empirically studied in phase 3.3, 3.4
    a. What context (OS, hardware, domain, etc.)

   b. Language mapping and constraints (C, C++ Java, and version/compiler)
6. Number of variants of the technique within each scope. (Quantitative)
7. The evaluation of the applicability of automation of the technique, which is a qualitative assessment. Measurements are ranging from (bad, slow and ineffective) on a floating scale to (good, fast, and perceived effective) (Qualitative)
8. Evaluation of the entire process (Qualitative)

We repeat that the applicability is especially important, since we know that techniques can be applied subjectively. We aim to define the techniques and their variants so they give measurable and objective results. This will also clarify what techniques, or what circumstances, are not measurable. We are using several ways to improve the validity of the result. We will use studies in a more laboratory setting, and treated the results through statistical analysis of our measurements. The evaluation was performed partly, in studies (see study 4 to 8 in chapter 6 to10) which it performed as the basis for setting up work in the area. Our aim was to further improve the validity of the result, by repeating the experiment with both controlled code samples and uncontrolled samples; unfortunately, we never pursued a fully controlled code sample. The technique was aimed to be repeated for untested code samples with different fault types injected. This attempt was particularly used in study 8 and 10. Finally, we aimed to manipulate the information on the number of faults injected (that could be expected to be found). The goal with this manipulation of information is to avoid self-fulfilling prophecies", i.e. if you know there are 5 seeded faults, you are looking for 5 (even if 10 were seeded!). We failed to conclude this approach.

We suggest that the code sample should be executed in different settings:

- No information is given on how many occurrences of faults exist, but information that the code is altered by fault injection.
- There is exact information on the total injected number of faults, but not on the type of faults, i.e. all faults cannot be found by the particular technique.
- The code sample is a part of a larger context; we will run a larger code selection in which there might be an uneven distribution of faults and no information on injected faults for

the other parts. This last setting is possible to explore when an automated technique is created.

In the last example, we are utilizing the automation technique on code that has not been a part of the manipulated sample (non-seeded). We will adapt the automation so that it can be used on code also from other similar systems. We aim to make the result general, so there are probably other approaches that we might need to explore to get good validity of the result. Even if this latter part cannot work as a controlled experiment, it will give additional information about the generality of our results, i.e. will strengthening or weakening our conclusion.

4.6 Discussion

The obvious plan was to follow the process as is. Attempts were made on a series of occasions (see study 3, 7 and 9). The first study, 3, was to establish a series of faults to inject in the code for comparison purposes. It would be clear to show what type of fault would propagate into a failure in what system – and then seeding a series of version of the systems, so different mixes could be created. Also the same "type" of fault could invoke different kind of failures in different systems.

The process did grow at parts grow to a halt in the first sub-process, where there still does not exist a sufficient fault taxonomy – and the failure propagation will always be within one and the same program, limiting generality and drawing conclusions on the technique. It would be possible to compare techniques only within the same program, and depending on the occurrence of each type of fault that would result into an observable failure (this was not concluded until finishing study 3 in Chapter 5). The second and third sub-process was used and is partially validated in studies 7, 8 and 10, see chapters 9, 10, and 12. Our initial aim is to evaluate our techniques and make our experiments only on a system running on an industrial platform, but we did use both industrial systems and open source systems, to ease our publication issues.

We planned to short-list the TDTs by selecting some simple area to demonstrate our framework, by using test design techniques (TDT)

based on input. We have been focusing on overlapping techniques, positive, negative, equivalence partitioning, boundary value analysis; fault injection and coverage with some use of random experiments were created. More automation driven techniques such as mutation testing and search based techniques have not been used.

We were looking into using the following scheme:

- TDT 1: Using "random" input [18][57] (one input, that could be expanded in many ways). This technique also means that we will identify test cases that require input, and will be an interface testing approach). This will then be expanding into two "random" inputs etc. But this was only partly evaluated.
- TDT 2: Using "Equivalence classes" [162][91][171], this means that there are at least n + 1 distinct values for every n boundaries that exist. These values will be selected randomly from the class.
- TDT 3: "Boundary value" techniques [18][21][131][216] that selects at least n + 2 specific values for every n boundary.
- TDT 4: Finally, we will use a heuristic method, called "Magic Values", in which we use knowledge of faults, creating a specific value class for potentially dangerous values. For example, using the data type integer, we would select 0, negative, large negative, floating values among others.
- TDT 5: Later we embraced a more detailed know-how around "Negative TDTs", a technique aimed at supplying invalid input, and use inappropriate (invalid) combinations together. This got expanded into several of Whittaker's well defined attacks.
- TDT 6: "Positive TDTs" which cater for valid input (and is a generic class, mostly made of "requirements" test – proof of concept).

All these techniques can be applied at several levels, and the main problem would be in automating not only the input, but also to identify (automatically) the test cases that require such input. We suspect that all these test design cases will be able to use similar techniques for this identification, but complementing the technique by adding different data types is more difficult.

In order to avoid prior knowledge of the system, we will use students. We want to avoid prior knowledge since we believe that it will be easier for experts to create "smart" test cases, instead of applying the

technique in a straightforward manner. Since most in-house testers have deep knowledge of how the system actually works, its weaknesses etc, they are more likely to use heuristic techniques than a student who is asked to apply a test technique rather monotonously. All students will evaluate the techniques and how and where they are to be applied. The focus will be on defining and understanding the different variants of application of a technique. Practical experience of how to use the techniques can speed up the manual application of the technique, which should also be taken into account. This means that the first technique the students apply will probably in general take longer (have less favorable times) than the next technique.

We believe that, even though designing the test cases might be a straightforward task for the student, determining if the result has caused problems (i.e. if the test case has failed or not), might need the expertise of experienced persons. Depending on the success of a pilot study, we will apply our framework on a larger scale and encourage others to use the framework for evaluating TDTs and their efficiency. To support comparisons in larger scale software samples, we aim to find a tool that could seed the code with different types of faults.
It is suggested by Ishoda [116] that the distribution of faults should be of the same number and distribution in the software experiment as in reality. We feel that although such requirements are important in the context of using fault injections to measure reliability; they are often hard to measure and enforce. The main goal in our research is to evaluate if one technique more efficiently finds the particular fault it is aimed to find, and with what coverage. A secondary goal is to investigate how efficient it is in finding faults of other types. Our evaluation will show both how TDTs relates to failures and indirectly to faults. In this context, Ishoda's argument is not applicable.

The samples used will have more faults injected than normal, and could thus be regarded as new and non-tested software made by a bad designer. We will have several samples seeded with different combinations of faults. It is important that we have fault seeded samples, but they should be "intelligent", in the way that the code compiles, and that it is not so fault infested that the code is not executing at all. Another problem is also in the unclear and coarse manner a technique (and its variants) is defined – *how* to apply the technique! We think it is important to clarify and define the testing technique in such a way that it can clearly be measured, and the

application of the technique can be replicated. We are interested in what types of faults that will impact the result. Therefore, thorough analyses of the types and classes as well as frequency of the faults are of interest. Even more interesting is how these faults propagate to visible failures during a test execution.

Chapter 5. Fault – Failure Classification

5.1 Summary

We have investigated a number of failures in a part of a subsystem of a large complex middleware system at Ericsson. These failures, are the grouped in classes, and debugged to their origin – the fault in the code. This study presents these failures and their frequencies, and isolates the failure classes that are caused by software faults. Our overall motivation is to create a controlled experiment by re-injecting known faults of different types in the code, to be able to learn more about efficiency and effectiveness of various TDTs [65].

We initially planned to use published information on commonly prevalent software faults, fault classes and their frequencies, but were unable to find sufficient information from existing literature. Since failures are related to software faults in complex and intricate manners, we believe that a realistic characterization of their correlation will be helpful in determining effective as well as cost-efficient testing strategies. Our objective is to understand how faults and failures manifest themselves, especially in the context of 'real' faults that occurred in commercial or industrial systems.

In this chapter, we describe the process used in our study for tracking faults from failures along with the details of failure data. We present the distribution and frequency of the failures along with some interesting findings unravelled while analyzing the origins of these failures. Firstly, though "simple" faults happen, together they account for only less than 10%. The majority of faults come from either missing code or path, or superfluous code, which are all faults that manifest themselves (propagate into an observable state) for the first time at integration/system level; not at component level. These faults are more frequent in the early versions of the software, and could very well be attributed to the difficulties in comprehending and specifying the context (and adjacent code) and its dependencies well enough, in a large complex system with time to market pressures. This exposes the limitations of component testing in such

complex systems and underlines the need for allocating more resources for higher level integration and system testing.

5.1.1 Context of this Study

We have investigated the interrelationship between software faults and failures. It is quite intricate to obtaining a meaningful characterization of it would definitely help the testing community in deciding on efficient and effective test strategies. In particular, this study was created as the first step of fulfilling our intention of our comparison framework, see chapter 6.3.2. Figure 3, the first sub-process, point 1.2. Towards this objective, we have investigated and classified failures observed in a large complex telecommunication industry middleware system during 2003- 2006.

The intention was to end up with a series of re-injectable faults that would propagate into visible failure in our real system, or to be able to obtain a fault classification of fault types, that could be considered generic enough that propagates in a series of system, and thus creates a realistic spread of failures that could be found by different TDTs and become sufficient basis for test technique comparisons. This investigation of fault and failures made us understand the complex relationships that exists between faults and failure, the lack of sufficient classifications of faults of different types, and the need to investigate further techniques for fault injection, mutation testing and a series of fault – language and system domain relationships. The final outcome also indicates how complicated faults might be, with a spread over several files for correction, and that in the real industrial system, the failure information is often kept, but the origin – the fault - and debugging paths are not as easily attained, especially when it is combined with other updates and re-designs of the same software. All this actions are normal procedure in a live system, and to improve research in the area, better information gathering for faults, improved fault injection techniques for real systems, better classifications and better tainting – or propagation tools must exists for software in real industrial environment – without burdening the development.

In practice, the result of this study, made us take a step back realizing that the work to conclude our ambition was far beyond the scope of this thesis, and a much more long term plan must be taken place to get a better juxtaposing of effectiveness of TDTs. Secondly, there is no information of

what faults are found during the design and development phase, since there is no recording taken place when developers find their own faults.

5.1.2 Design - Research Method

The research method set up for this research was by first sing a series of classification existing for different specific aspects of software, and then create a first attempt. The selection methods are detailed described below in chapter 5.2. The classification scheme was then reviewed and refined, and the definitions clarified. We do not feel the classifications are sufficient, but they pose a step forward, and enabled us to look at distributions in a somewhat sampled selection.

5.1.3 Validity Threats

This is a study, which has low control over the environment, we have had some control over the measurements, but they were selected on the basis of possibility to gather (based on the labeling feature), which was random and outside our control. The replication is probably low of the experiment itself, but it should be possible to take any known failure/fault classes and do the same classification with a different outcome, depending on the software, process, organization and situation.

The main argument favoring our study is that, all failures and thus faults are from a live real system, and it is representative of the faults found and fixed, even if it is a not so huge sample. One possible validity threat is the selection of data is created based on the labeling function. The obvious way to perform a study on distribution of failures into fault classes would be to look at all faults, by comparing the difference between code versions. This is not a viable approach, since in systems, fault corrections, changes by adding new and modified code are mixed. Secondly, we have only focused on one small part of the system, since the main purpose was to identify the faults that we could make a copy of and re-inject the faults to use in controlled experiments. We investigated that for every reported failure, finding the actual fault, which represents the correction in the code, is still not completely accurate, since there is not a one to one relationship and this creates a problem in how to classify a fault. This is why we have shown two values, one based on the failure data reported, and one based on the

adjusted software fault correction possible to re-inject in the code. The selection was made out of convenience, to find and create a sample to reason about. The failures corrections (labels) selection came from a wide community, of 65 designers from many countries and collected over a long time period and over many versions and changes in the system (3 years). We draw the conclusion that this fact lessens the internal threat, since the bias of interpretation has less impact, but cannot be disregarded.

Conclusion validity relates to subject selection, data collection, measurement reliability, and the validity of the statistical tests. These issues have been addressed in the design of the experiment, and we believe there is a disturbance in the data collection and measure reliability. We have used a nominal scale to classify our Trouble Reports. The classification is rather simple, which could be indicating problems with the internal validity. Classification into another system will yield a different result. Since our distribution is done with little insight of the software and no insight of history, process and organization and by and external party (thesis worker) the bias is minimized. There is no researcher bias put into the investigation, since it was done by a third party, and no guidance was given to what set of faults should be investigated, how the distribution and classification should be used or chosen. Thus, the researcher, who is familiar with some aspects of history, organization, process and software, which could indicate a bias, and a threat to the validity, have made the conclusions and verified the result.

Discussing the generalization of the result, we must look at external threats. The faults selected are on one particular product, but the nature of the product is such that it could probably be representative for lots of industrial software. We cannot conclude that that the result is generic since the distribution is dependent on organization, process, quality awareness, and a lot of other factors. However, earlier studies with similar results [20] supports that for these types of systems the results could be generic and not an isolated result. We do think this is one example of a typical industry software fault distribution. Using our conclusions for any system might be pre-mature, but we suggest that this information can serve as an indication for complex middleware systems, operating systems, and similar large complex systems. All data is still available to pursue further studies.

5.1.4 Contribution

The questions we primarily try to address of which we have answered in this study are:

- What are the real and important faults in software that have propagated to failures, and subsequently fixed?
- What is the distribution of the failures and faults into various classes? (This classification will allow us to re-inject faults of the same type in the software, thereby providing a basis for our planned evaluation of TDTs.)
- Is there any other specific pattern of faults and failures that would guide us into understanding the software process better?

The contribution is that we *defined a classification of faults and provided data for specific distribution*. In addition we could see patterns of faults and how they are spread over many files, not only residing in one place. We have also been showing that not only software faults causes problems, but also other factors are strongly contributing. Details of the contributions and findings of this study can be found in 5.4.2 where the use of the taxonomy and a replication in another system can be found in [81][82].

5.2 Introduction

Most software industries do not pay enough attention to understand the typical faults that exists in their software. These industries collect every anomaly and complaint, from both verification teams and customers, but seldom faults and failures found by designers. Hence, failures are collected from the later stages in the software process, saved in databases, classified based on priority, status of management (e.g. analyzed, fixed, tested, approved) and classified based on organization or software sub-system where the actual fault is believed to exist. Occasionally deeper analysis and classifications are done and root-cause analysis (RCA) is performed, especially when major incidents involving customer complaints occur. Most classifications [37][196][111][46] end prematurely by defining the failure on too high level to understand what software fault caused the failure. We suspect that this is the case in most industry and commercially available software, with the exception of safety critical software.

In this study, we consider the software in a typical telecom system. We have not looked at all the reported failures for the entire node, but focused on one particular part of the software, which we consider to be a typical representative of telecom middleware. From 4769 reported failures, we have selected a sample of 362 failures, which can be considered to be important since they were all corrected. This data has been collected during a period of three years based on 65 different designers and software integrators across the world. We chose these failures since their labels in the configuration management system made it easier to locate the corresponding software fault, compared to repeating the tedious troubleshooting and debugging of the system, which would otherwise have been required.

Our focus is on software faults, but our study has shown that just less than half of the reported failures are not a direct consequence of a software fault that can possibly be re-injected in the code. Instead, failures relate to a variety of problems, e.g. hardware, third party products, process issues, organization, and management issues. We decided to keep all information to give a better perspective for researchers trying to understand problems in the software industry, and better explain them as a part of our study.

5.2.1 Related Work

Our purpose is to investigate software TDTs is described in our position paper [65]. We noticed that most test technique investigations used small code samples, often with very few faults injected [128][6]. The faults used as the basis in test technique investigations are often invented and "simple" or made to prove a specific point [137]. This did not match our experience with faults in software for complex systems. Even if there exist attempts to create better faulty programs to use for TDTs research [105] [3][79][205], they still do not contain enough data from industry, and are relatively small compared to our complex middleware. Therefore, we argue that the early stages fault investigations for test technique research [13][162][98] needs to be updated. Andrews et al.[3], share a similar goal but uses a different approach. They compared mutant generated faults with hand-seeded real faults, and concluded that the faults were different in nature, but shows statistical promise. Analyzing our result in contrast with theirs, it becomes evident that the type, nature of the fault and fault class, and its distribution *is not sufficiently explored* to draw strong conclusions about TDTs. We

tried to find examples of classes and types of code faults to re-inject, but did not find any good list to use, instead we found several papers investigating real faults (and failures) classified for different purposes [12][35][46][51] [72][95][97][137][160][168][173][174][176][219] and not distinguishing faults from failure or cause. In particular, for the more commonly used Orthogonal Defect Classification (ODC) [35], we concluded that the classes are classifying failures, and not faults, e.g. an interface failure (which is an ODC class) could be caused by several software code faults. Thus, these classifications are insufficient to support us in our aim. DeMillo and Marthur [51] have made an attempt to classify real software defects by automatic means.

We have used these fault and failure classifications as inspiration to our classification and we will discuss them in the section 5.4.1. Rather than adopting, we strongly propose that different software domains have different sets of fault and fault distributions, depending on organizations, languages, development and testing methods, as well as the ways of measurement. Conclusions on TDTs should be based on first creating a thorough fault analysis particular to the domain, but with known methods. This will provide better understanding of which typical failures and related faults that are relevant for this particular software. We realize that the gap between research and industry is wide [18], but we hope to close this gap by doing controlled research on commercial software in an isolated environment. Offutt and Huffman-Hayes [168] discuss in the semantics of a fault. This is interesting research, since it implies that faults have a variety of impact, depending on the fault. Our research shows that some types of faults that affect the software are a combination of faults, and are definitely involving more than one file, and more than one entry in a file. There is a danger in inserting only single semantic faults even if they are dominating. Single semantic fault injection is the predominant way of injecting faults (and mutations).

One fault can propagate into many different symptoms (which is one of the explanations to the high number of duplications). One fault might propagate and behave differently, depending on how “complicated” it is. Hyonsook and Elbaum [105] have with the work on SIR framework, used files from industry (SPACE and Siemens program [205][111]), but also let experienced designers deliberately insert faults (also used by [3]). There is no way of telling if these faults are representative of common faults in any system, or if they are too simplistic in nature. Our initial reaction when

analyzing failures have been that faults that dominate are much more complicated in nature than plain logical or computational faults, which do occur, but not as frequent. Ishoda [116] argues that basing research by capture- recapture (inserting faults, and then estimating how many of them are found) is not a sufficient technique for reliability (*and test evaluations*) analysis, and that correct frequency and type of faults must be known for the software in question. We support this argument, which also lead us to investigate our own frequency and type, to understand the characteristics of our particular type of software. Ostrand, Weuyker, and Bell [174] work in the same domain as us, large middleware systems with similar problems. They have not focused on the fault type in itself, but on occurrences and location, which is similar to our approach. Since they have classified based on their MR (similar to Ericsson's TR), we also assume (but have not verified) that their data suffer from the same problem as our, what is reported (failure) and the connection to the actual fault in some code file needs a much further analysis. Yet, they report interesting results that seem to match our experiences: Most of the faults reside in (or are reported on) 20% of the software, i.e. 80% of the software is more or less fault-free. Furthermore, their distribution seems to match ours, with over 50% of the failures related to missing or spurious code. A comparison with our figures shows that their distribution and frequency is very similar to ours, even if it is done more than 20 years ago. We will discuss this further in our last section: Discussions and conclusions.

The key problem we would like to address is that there is no recent industry data available for research purposes. In addition, people who measure are often using too high level classifications [37][111] to be useful for our purpose. We have also studied bug taxonomies [19], but conclude that they mix cause, fault and failure, and are seldom providing obvious support for re-injecting faults, even if they give valuable information about failures.

Our main conclusion is that it is important to regularly collect and report findings of this nature from real industrial and commercially used systems to keep information in tune with development approaches, software and faults. We also assume that there is a large diversity of the frequency of faults, depending on what type of system and domain they reside in. We also suggest that within a domain – or type of system (e.g. a large complex operating system middleware with partially proprietary hardware) it is possible to find similar structures across the entire domain, which could

indicate that the results are not limited to only e.g. telecommunication middleware systems.

5.3 Study Process and Data Selection

The process followed in our study is described in Fig. 5.1. We started by selecting the Trouble Reports (TRs) for which a link to the corresponding code exist; using a script that automatically linked the fault id into the corrected code as a comment. Then we compared the corrected code with the original version of the code, to identify areas in the code where the fault could reside. This is a non-trivial task, since enhancements and improvements to the code are mixed with corrections. We then classified each Trouble Report into one of the chosen failure classes.

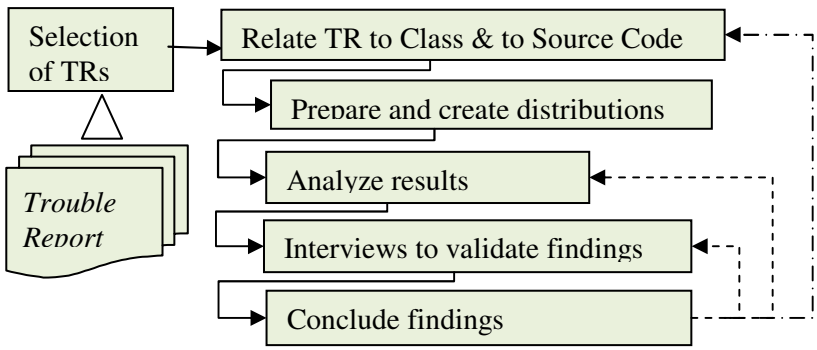


Figure 5.1 Overview of the classification process.

The distribution of failures was then analyzed, followed by some interviews of the designers, with the aim to validate our findings. The component size was approximately 180-200 000 lines of non-commented code (measured over the three year period). For the entire middleware system, there were 4769 TRs reported and of these we have considered 1191, indicating that it is a central part of the software. From these reported TRs (and also, from the complete set of 4769 TRs), some of the TRs have been analyzed to originate from faults elsewhere, or require corrections in two places.

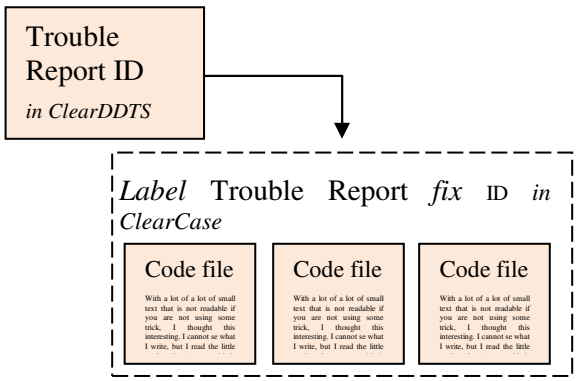


Figure 5.2 Failure (Trouble Report) – fault/fix relation, where the label points out involved files

We must understand that not all of the 1191 TRs lead to a correction. There are 181 TRs reported directly on this component's code during this period. Our number is higher (362) which indicates that TRs that reside elsewhere (are reported on other places in the code) affects this code. The TRs within this target are all using a particular labeling function that makes it possible to trace the TR to the actual file (see Fig. 5.2).

How this subset (of labeling) is related to the 1191 TRs we have not been able to pinpoint, since it would have required serious data mining, taking all change requests in to account and other factors. From the entire set of 362 TRs, we now withdraw all faults that are not software related, e.g. duplications and wrong usage; to come up with our set of 295 unique failures, and of them only 170 (204 adjusted[3]) are software failures. This means that approximate 14% (17% adjusted) of the reported failures (1191 TRs) at least are real software code faults!

Little or no general tool-support exists to trace, categorize and get support in tracking down the fault causing a failure. Currently this has for non-trivial cases to be handled by manual work through code inspection. Manual comparisons of two versions of the source code is often the used

[3] Our investigation showed that for many reported failures, the actual fault (represented by the corrections in the code) is not unique and failures and faults do not have a one to one relationship. Therefore, we have adjusted our figure, by adding all software faults that are contributing to the cause of a failure.

method to extract the actual difference, but this is not enough, since enhances to the software and other modifications must be excluded and the fault origin or origins must be isolated. The problem is non-trivial in systems with a multitude of code branches, since it amounts to know where to look without having to scan through a multitude of files and to separate fault corrections from any other code modifications, improvements and change requests.

This is the true complication of relating a "unique" failure to a "unique" fault since they do not have a one to one relationship. What even more complicated our classification was that the original designers (and trouble-shooters) were not available to ask of the real origin of the fault. Doing a complete trace could take us between 3 days to two weeks, which explains why this route to identify and gather faults from failures is seldom taken for this type of research. With the labeling function, the connection between the TR-system id and the actual place in the code where the fault could reside could be made, and capturing the correct code became a task of analysis between one hour and 2 days, when done by a person not familiar with the software. We consider this a great improvement.

5.4 Identified Failure Distributions

This section presents the fault classification, the distribution of failures observed in our study together with the distribution of the software faults over number of affected code files.

5.4.1 Fault Classification

In our classification we use the following classes of faults:

- *Language Pitfall* are faults that are specific to the programming languages used, e.g. pointer being null, pointer pointing to an invalid address, valid address but pointing at garbage. E.g. arrays is a common type of data structure accessed with index, and if the value falls outside the boundary of the array then the accessed element is unknown, or data might be modified that shouldn't. In addition, overflow and underflow are categorized in this class.

- *Computational/Logical faults* are faults inside a computational expression. A logical fault is similar to a computational fault, except that it is related to a logical expression.
- *Fault of omission* is when a fault happens due to missing functionality, i.e. the code that is necessary is missing. E.g. a part, an entire statement or a block of statements missing can be classified as faults of omission, which means missing either of the following: function call, control flow path, computational or logical expression.
- *Spurious faults* are similar to Faults of omission, but in this case there is "too much code", and the correction is to remove one or several statements.
- *Function faults* are faults, such as calling the function with the wrong parameter or calling the wrong function.
- *Data faults* including faults of several types: Primitive data faults, which include defining a variable to the wrong primitive data type, e.g. integer to be unsigned but should not have been; Composite data fault e.g. structure (in C or C++); initialization fault and assignment fault.
- *Resource faults* are faults that deal with some kind of resource, such as memory or a time, therefore this class handles faults from allocation, de-allocation, race-conditions, time issues (dead-lock) and space (memory, stack etc).
- *Static code fault class is* code that does not change after compilation of software or after the first execution, when using an interpreted language. This code is e.g. source code or configuration files which the software when executing uses.
- *Third party faults* are faults in software for which we do not have access to the source code and hence cannot correct ourselves.
- *Hardware faults* and *Documentation faults* are self-explanatory classes, and do not relate to software.

We have grouped the above into four groups, viz., code, process, configuration management (CM) and other.

5.5 Fault Distribution

The selected of 362 trouble reports, were distributed as explained in Table 1 below, where faults of omissions topped the list. The second largest class of Trouble Reports is *Duplicates*. One failure out of three non-software

related failures reported is a duplicate. What is interesting is that these duplicates were not identified as duplicates until they were corrected in the code. This means that the TRs were individually decided to be fixed, and assumed to be of different fault origin, since they showed different behavior and were viewed as different failures. Otherwise, these failures would have been omitted before ordering them to be fixed. The high number of duplication of failures is also related to the way testing is done on this complex system, and also to the type of software (which most other parts of the software were dependent upon). This means that many failures in this particular software will be found and reported by several persons.

Table 5.1 Failure distributions into classes and their frequency. Second column is the *adjusted* value translating TR to fault classes

Group	Fault/Failure Class	Failures	Code Fault *adjusted*
Code	Faults of omission	73	78
Process	Duplicate of TR	67	-
Code	Data fault	22	26
CM	No files associated	21	-
Process	Change requests	21	-
Code	Static code fault	20	21
Code	Spurious faults	17	17
CM	SCM	15	-
Code	Resource fault	13	13
Code	Computational/ Logical fault	11	28
Code	Function fault	11	13
Process	Fault not fixed	9	-
CM	Compile time fault	8	-
Other	Third party fault	5	-
Other	Documentation fault	4	-
Code	Language Pitfall	3	8
Other	Hardware fault	2	-
Process	Not a fault	1	-
Code/Other	Too difficult to classify	39	-
	Sum	**362**	**204**

Another aspect is that failure reports are symptoms, and they might not appear in the same way (and be explained similarly) by two different persons. This problem is recently remedied, by enhancing the pre-test on the entire system before release (much as a result of the insight gained by this study).

Disregarding duplicates of failures leading to the same fault, we would focus on the 295 unique failures. However, the classification needs adjustment, in relation to how some failures are reported. For example, if a fault needs two corrections to be fixed, and there are duplicate TRs, there is no way of telling that one TR correction is assigned to one or the other fault correction, and the "duplicate" vice versa. We see this as a disturbance of the data, but have reported the adjustment for the fault classes in software. The adjusted values will if re-inserted cause failures. We have started further investigations to understand the nuances of how faults actually infect the software, and when and how visible they are.

The next interesting class of failures was "too difficult to classify" containing 39 of the failures. The reason is that it is impossible to pinpoint the exact difference, since the entire unit or file was re-designed, and large portions of the code rewritten. This also makes it difficult to pinpoint if the change was only due to the failure, or due to other factors such as updates, expansions, new features etc. We decided that for our purposes is not worth the effort to sort this out; instead we just conclude that 13% of the unique failures lead to a major change in the system. Only 8 (of 19) classes can be considered in our investigation, since these are the ones directly related to software faults. This is only 170 of 362 reported failures (47%), or 170 out of 295 unique failures (58%). This means that as much as 53% can be dismissed due to problems that are either indirect, processor or system related, or disturbance in data and that these failures do not uniquely originate from the software. Third party faults are software faults, but they cannot be traced into code, since the source code is not always available to us, and must be corrected by another party. Another category is how the system is built and integrated, including software configuration management (SCM) failures, compile time failures (during the build) and the category of "no files associated". These failures are strongly related to the fact that a major change of build system and product structure happened during the period of data collection, which explains why it constitutes 1 out of 5 non-software related faults.

In Table 5.2, we present distribution of faults together with the number of files which had to be updated to correct the faults. Table 5.2 Distribution of adjusted faults into number of files

Code fault class	**Faults**	**%**	**Number of files**										
			1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**11**	**25**
Faults of omission	78	38.2	50	12	8	4	2	1		1			
Computational Logical fault	28	13.7	24	2		2							
Data fault	26	12.7	20	3	1		2						
Static Code Fault	21	10.3	4	7	2	1	1	3	1	1			1
Spurious faults	17	8.3	11	2	1	1	1					1	
Resource fault	13	6.4	8	5									
Function fault	13	6.4	11					1			1		
Language Pitfall	8	3.9	6	1	1								
Summation	**204**	**99.9**	**134**	**32**	**13**	**8**	**6**	**5**	**1**	**2**	**1**	**1**	**1**

The failures found in the study, can involve between 1 to 64 files. The software faults can be distributed between 1 to 25 files. Analyzing this data shows the majority (66%) are from 1 file, but e.g. that faults of omission involve more than one file for 36%. We have not looked at the details, such as the location of the faults within the file, which is demands a more in depth investigation, or if the files are all "owned" by the same designer or not.

Our result was not what we expected, since we have had the assumption that e.g. more than 4% would be language pitfalls. This is why we felt reporting these findings would aid others in understanding more about the nature of faults. Our most important findings can be summarized as follows:

- Most of the faults were faults of omission or "missing path", meaning, not until execution on higher integration and system levels the lack of code was noted and the failure visible. A designer could clearly not find this, and the cause is most likely insufficient specifications available on lower level or lack of knowledge of the context of the code. A related fault class is spurious faults, where too much code is written, which might be overlapping or creating the problem.

- More than 34% of the corrections involve more than one file and the maximum of involved files for a correction is 25 files – the conclusion is that these faults are not possible to find on component level, and even 100% code-coverage on component level would not reveal the failure.
- Faults are much more complex than simple mistakes; often complicated logic confuses the developer. The semantics of faults are complex.
- Resource faults are not as complicated as e.g. static code faults when it comes to distribution of location.

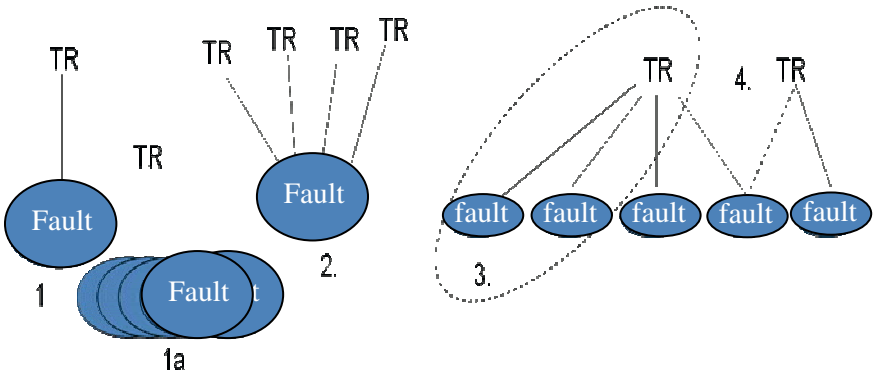


Figure 5.3 Fault – failure (TR) possible relationships is not only one to one

5.6 Discussions and Conclusions

5.6.1 Problems with this Categorization

For natural reasons, a lot of the designer faults are found by designers themselves and are not registered. Depending on industry, designers are usually responsible for a large amount of the lower-level testing (unit, component, and lower level integration, and even some functional testing). Therefore, data on failures originates from independent test at different

integration points, (levels), and from customers reports. We intend to investigate what faults and failures the designers themselves find.

Since we have noticed fault classifications often become failure or cause classifications, we aim to do a further study and make a fault classification (bug taxonomy) that are more useful for fault injection purposes. We believe we have already hinted on a structure (defining our classes and sub-classes), but feel we need to investigate and possibly expand these further. We also need to provide more clear rules for how to classify faults, so they cannot be classified in different classes depending on how the fault is interpreted. The large number of existing fault classifications will also be juxtaposed in such a new classification.

We would like to investigate automatic ways to classify legacy software, but that is not our primary concern, and we invite other researchers into a discussion about the feasibility of automatic classifications, as we have seen a published example of this [51].

Our primary concern is to prepare code in many different versions, with real faults injected. These fault-injected code samples are intended to be used for evaluating TDTs in a controlled manner. We aim to inject faults into a system to minimize bias for one test technique over another. This require us to have a clear classification (with relation to the test technique), and there must be a variety of faults that behave (propagate) and is viewed differently, if we are to determine where a test technique would be more efficient. Our interest to find a set of faults that through execution behaved in different ways, and cause different failures in the software. The faults we have found, isolated, classified, and understood are to be re-injected in the code for better comprehension of how each one of them behaves when propagating to failures, as opposed to debugging. Of course, the variety of faults is more interesting, and we are to explore mutation techniques as a complement, and study how they behave and how their fault semantics could look like.

5.6.2 Findings based on Data

We now return to the questions posed in the introduction and present our findings based on the collected data.

1. *What are real and important faults in software that have propagated to failures, and are fixed?*

The answer is clearly that faults of omission (38,3% of software faults), together with spurious faults (8,3 % of all software faults), shows that faults related to unclear specifications dominates among the real software faults found in the considered sub-system in a large complex middleware system. The main conclusion and contribution is the fact that the individual designer at component test level do not find these type of faults, and that they must be found at later stages in testing. It underlines how difficult it is for a designer to understand, define, specify and implement code in an environment of complexity, since not enough knowledge of context available. This is supported by looking at distribution of the faults over files, where 36% of these two classes of faults spans over more than one file.

2. *What is the distribution of failures and faults into classes?*

Our observations were as follows:

- 53% of all failures decided to be corrected are not relating to the software and are not possible to re-inject into the software
- Faults of omission (lack of code) is the dominating class among the software faults (38.3%)
- Computational/Logical (13.7%) and Data faults (12.7%) followed by Static Code faults (10.3%) are the next largest groups.
- Language pitfalls are only contributing with 4% of the software fault distribution

3. *Is there any other specific pattern of the faults and failures that would guide us into understanding the software process better?*

- We have found that as much as 19% of all faults are duplicates that still remain to be corrected (even after management has taken out duplicates). This shows that software failures are often expressed differently and not identified as duplicates until code is corrected.
- Software configuration management and build related faults together are contributing with as much as 15% of the failures.
- Conventional faults in software (computational/logical and functional faults, language pitfalls and static code faults) are only 39.6% of the software faults in this complex middleware system.

5.6.3 Discussion on our Findings

We think that these findings suggest where effort and cost should be placed. We know for a fact that this software had a targeted component test improvement (unit level) during the year 2005, which greatly improved the quality. We know that from the period 2003-2004 most of the trouble reports came from outside this software organization (applications, customers), but after the improvement most of the trouble reports originated from within the organization. We also know that the testing of these products have improved between 2003 and 2006, with only one test level in 2003, and now with more than 4 test levels. We can also see this reflecting in the number of duplications going down. We have noticed that viewing Trouble Reports (failures) over time, provides us information that many of the changes induced by the trouble reports that we considered were both re-designs for change and expansions purposes and for fixing bad design, and these problems were more common earlier in the development than in the later stages of development. We suggest that changing software configuration and build system will have a great impact on the code (since our study indicated as much as 15% of faults are due to SCM). We think there is much information to be utilized from this study. We do believe it is humanly impossible to understand the entire system, and even if knowing context and teaching about it, this will only remedy a part of the problem, and that unit and component testing alone have no chance of finding a majority of these faults, which is of course already shown and evident. This study strengthens that evidence. Even if we strongly believe component testing is essential for complex systems that need to be robust, we must do testing on many other levels in the system to understand where the important faults hide. Testing is a support to the designer.

We were surprised that the fault categories that are the target of many static analysis tools had so low frequency, which leads us to suggesting a more cost–efficient way might be to work on specifications, understanding the software context, and test-set up. We believe that since the system is build in a “fail-safe” way with the aim to minimize impact of anything going wrong (by duplication of hardware, protocol resending, restarting etc) the impact of simple singular faults are often hidden or dormant. We believe that the improvement on component test are not reported in the same way in the trouble report system, since designers correct their own mistakes when they encounter them, rather than report them, which puts a hidden

figure on these types of faults. Most of the component test faults (that is found by e.g. static analysis tools and component test) are not as visible in this study, but still do exist.

We have concluded why there is such a lack of information on code faults – and how good it is to do this analysis and really understand the information, since it gives Ericsson guidance on where efforts of improvements should be targeted. We understand the difficulty to gather this information if traceability of the code is not directly possible from the reported failure. The strength in our study is that it is unbiased to the code, and based solely on applying clear classification rules. We have of course encountered problems with classifications, how distinct the classes are, and how classifications should be applied. This work provides a milestone in this research. It shows the difficulty and the importance of understanding fault- failure relations much better, and that classifications and better data is needed. Not only is debugging and tracing faults important, but the propagation of faults into failures equally.

Chapter 6. Improving Test Data in Automated Test Suites

6.1 Summary

Applying a TDT in practice is not always straight forward. This chapter describes a study performed in an industrial setting. The aim was to improve the effectiveness of the functional test cases to find more faults with limited extra effort. All test cases are based on test specifications and expressed in a test suite that is automated. We expanded existing test suites by simply adding more structured input data, using straight forward TDTs, thus explored special values and boundary value analysis to improve the test suite.

Through thorough review of the test specifications and test cases, we concluded that the test cases mainly represented one normal, valid, positive test case, where input was coded as the only input for the test case. No particular TDT had been used. In our study, we are trying to add new input selection, based on known TDTs with the aim to expose failures in the software. We created a list of suggestions of how to select input based on known TDTs. We were curious if this redesign of the cases (applying structured techniques) would really lead to more faults being found, and how many. We applied the technique twice, using different software for our experiments. The first experiment found no new failures in the software, and the second found a substantial amount of failures. This study will describe these two experiments, our results and our insights in applying TDTs.

6.1.1 Context of this Study

Our main goal with our research was to find a way to compare TDTs. With our lack of success in re-injecting faults into controlled software, we decided to focus directly on the TDTs and the industrial software. It should

be possible to add techniques that found more faults, and the payback would not only be to get better understanding of how to conduct an industrial study, but also create a motivation for the participating industry, by finding actual faults in their test suite. By reviewing the existing industrial automated test suite, and test specification we hoped that through identify fallacies, we could provide justified evidence that could be directly linked to the usage of established TDTs.

This study enlightened us of a series of problems when dealing with industrial case studies. If, as in this case, there is no direct involvement during the entire process by the researcher, there is a series of lack of data, information and details about the study that can or could entirely jeopardize the validity of the result. And as we found out during this study, is that we needed to re-apply the same initial approach – but now we more detailed know-how of the technique on new software. This did unfortunately also jeopardize the result, since two main factors was changed at the same time, since not only did the way the TDT change how it was applied but also the system under test. Because of this, the data concluding makes it hard to make a conclusive result in regards to the TDTs. The most surprising element to this study is how meticulous one must be in how the TDTs are applied, which made us focus even more how important it was to understand the actual application of the TDT.

6.1.2 Design - Research Method

The intention was to review the automated suite, improve the test data selection based on techniques, and then re-evaluate the improvement made and thus measure the faults found. The test specifications written were first reviewed by the researcher to identify possible TDTs used or that could be used. The test specifications could not be classified to what different techniques that had been used, since they did only contain one “accidental” technique, meaning – they demonstrated how the software would work in one case, the normal case. This did not provide any interesting data.

Then we decided to apply a series of “trivial” techniques on one automated test suite, in a very simple form. The industry was to choose the software under test, and find a suitable candidate. They selected a well established test suite – which definitely would challenge the ability of applying techniques, if they were all done in this manner. It was identified that the

automated scripts had hardcoded one data value, instead of iterating over a series of input exercising the software under test. We then suggested suitable changes to vary input. After the initial trial, which did not produce any data, since nothing was found, no figures of how many new test was created etc, was non-existent.

The industry decided that the system used should be changed. During this discussion, the main-problem, the wrong application of the technique became clear, since the researcher insisted that at least something should in theory be found. In hindsight, changing system made the data collected useless, since the ability to compare two sets of measured data was just not there. Even if the newer system did provide some data, there were not sufficient information on how data was collected, and the insight in the how makes one question the result collected.

6.1.3 Validity Threats

There are many threats to this study. Not only is there a lack of information regarding how the data collection happened, but enough detail to trust the results. There is no researcher bias on the data collection. The most important threat to validity is that the sample of just 100 input values is too small to draw any conclusions, no statistical significance is possible. The construct validity shows the target software has a great impact on the result, and is crucial to the result. More impact on the construct validity is the list of suggested improvement supplied.

Another threat to construct validity is that the list of what to do is geared for testers to follow a systematic approach, but instead the researcher bias is low, since the list was not followed systematically. This unfortunately also lower the value of the experiment, but indicates that when testers are involved and intelligence can be applied it will. We mean the selection of input will always be guided by something that makes some sense, when applied by a tester. The testers were not particular aware that their results was a part of a study.

6.1.4 Contribution

The contribution is the insights in applying TDTs can play havoc, and thus giving us a new insight in the TDTs. The suggested simple TDTs can

contribute to testing and thus how important it is to have varied input even if a fully automated system, thus this is a technique that aids in achieving an effective expansion of the automated suite.

6.1.5 Conclusions

When conducting this study we realized the difficulty of using these rather simple techniques, and that the set-up of the experiment is vital. This indicates that when and how you apply a test technique or approach is very critical.

6.2 Introduction

Applying a TDT in practice is not always straight forward. This chapter describes a study performed in an industrial setting. The aim was to improve the effectiveness of the functional test cases to find more faults with limited extra effort. All test cases are based on test specifications and expressed in a test suite that is automated. We have explored special values and boundary value analysis to improve the test suite. When conducting this study we realized the difficulty of using these rather simple techniques, and that the set-up of the experiment is vital. This indicates that when and how you apply a test technique or approach is very critical. The contribution is the insights in applying TDTs and achieving an effective expansion of the automated suite.

We had reviewed a series of test specification and their resulting test cases and noticed that the input selected is in most cases demonstrated the normal behavior of the system functions. This means that the test cases were mainly created for “requirement testing”. The technique used are sometimes called exactly that, “requirement TDT”, and could be described as the normal or valid test cases, where an input is selected that would not exercise any failure case or trigger any fault handling of the software. This is also called a positive TDT, where the negative test case does focus on triggering failure cases or fault handing of the software. Testing negative cases are always important for a robust test of the software showing flaws in the design.

Basili and Selby did already 1987 compare boundary value analysis as a functional test technique in different settings. Reid clearly showed that the value of testing with boundary values [182] is far beyond that of equivalence partitioning and random testing. This technique is defined with examples in BS7925-2[30]. Using maximal and minimal values outside the boundary can be defined as having a negative TDT [127].

6.2.1 Applying TDTs in Industrial System

The system in this study is an industrial system. We asked ourselves, how can you vary input to an automated suite of test cases with little effort? This is our main research question. The interesting part of this system is that all test cases were already fully automated, but with only one input – hard coded in the test case. In this particular case, we wanted more evidence to really be able to claim that the TDT – basically – variations of input, is a viable approach, meaning it would be worth attempting using different TDTs, and that this could contribute to finding important faults in the system. Without this hope of actual improvement, there is little interest from the industry to participate in this work, yet alone, it is not worth the effort to redesign the system in this manner. It is important to show some simple techniques, and how these contribute to better choice of input data that can to expose failures in software.

The idea came out of the initial analysis of the functional test specifications, on which the automated scripts were built. It was obvious that no systematic test technique had been applied, but a multitude of them in a mix. This led to the creation of a list of improvement and suggested where we wanted to investigate the efficiency and effectiveness of the TDTs a bit further.

We realized that it requires a large effort to rewrite all test cases and insert variables instead of hard coded input values, but saw this as possible improvement. In addition to this work, another redesign of the test scripts should be done, for the test scripts to be able to accept input and at the same time have an associated expected test result with this input. Both of these should be sent as an input if you want to retain the fully automated approach, using a variable input – but same execution. We calculated that the effort would be substantial. Separating input data from the execution might not be the best way to approach an already existing test suite, but is a

good suggestion if building automating a new test case execution from the beginning.

We thus continued with our research question, that intelligently selected input, based on established techniques, would yield a far more focused effort. We decided to create a list of suggestions on how to vary – on a general level, the different test approaches. This list looked like this:

1. Maximal and minimal values if possible, otherwise a very high and/or a very low value
2. Magic numbers -1,0,1 for values, and large numbers changing to exponential notation
3. "ASCII"-table (numbers), and some magic "numbers" e.g. tab, 1, "null", "space" empty set etc.
4. Any other selected value that makes sense in a set of values to test failure

In addition, the aim was to change the order of the scripts, when they were not obviously consecutive. Action could include mixing, i.e. to start test scripts, make other tests in between and then continue with next script, or just swap the order of execution for two or more test scripts. This last part of the study was never concluded to the researcher's knowledge.

6.3 First Attempt at Industry Data Collection

Testers performed the study as normal work in an industrial setting; hence this experiment is just ordinary work for them. Only very few persons were involved in the trial. The aim was to see if this simple approach – using some technique to better select input to the test cases, without much effort could yield any improvement in the effectiveness of the test cases.
The testers selected software that was familiar, and that had not yielded any faults for several versions. The main variation was introducing boundary values, but a whole list of variations of data was supposedly to be created. The experiment was implemented and the results came back. To our astonishment, not a single failure could be found. The software had not yielded faults for several versions, and we concluded it was very well tested software available to customers. Since the tested software has been on the market for while, we assumed that most failures must have been

found in later stages than this system test level. We discussed to repeat the experiment on earlier versions of the software, but a busy industrial testing team has not really the time for more experiments if there is no direct motivation. This track still needs to be explored. Secondly, since we felt it was amazing that the software was so well tested, we started to investigate exactly how the data was varied. It turned out that where we aimed to check the boundary values, the tester explained that he had simply picked the boundary in itself only, and not three values for each boundary (which is a necessity for checking a < ,>= relation.

Well, did the experiment fell flat because of not knowing how to apply a well known technique? This led us more to look at how is the technique applied, and what does this mean. We had to double check if this is a common misunderstanding of the technique. We have now checked with three different test audiences at different occasions (of testers from many companies) and - to our surprise, a majority in every group believe boundary value testing is picking only the boundary, not that you need to test the value before and after the boundary too. Not a statistical significant test, but definitely enough evidence to see this as a major risk for the applicability of the test technique. This misunderstanding might have been the actual explanation of the result from the first experiment. But instead to redo the experiment, the industry/testers had already decided that this software was not interesting. We decided to repeat the work and add the new additional values. Unfortunately we did not insist on the same system, but just the same approach – and did now aim for the same approach on newer software. This new software was much more untested than the original study. There were many reasons for this change of selection:

- Newer software (more untested and unused) is more likely to contain faults leading to failures.
- It is more motivating for testers to find failures, which motivates working further on these additional test cases.
- If we were to find failures, this would build a case of allowing further work (and research) in this area.

6.4 Second Attempt to Collect Industry Data

The setup of this study was principally the same; the difference is that now we could now assume that the actual boundary value analysis was performed correctly. The targeted subsystem in this experiment had been selected from an area that was new, and had not been long in the field. In detail, we have compared the hard-coded automation input and added test cases by duplication, and added new input according to the basic technique – and list provided.

6.4.1 Input to Experiment

The selection of data is to a high degree based on the technique, but not fully, since many inputs do not have a clear boundary. Another reason is that the testers have not pursued all boundary values. If the test technique was applied more stringently, there might be more failures to expose.
Selecting correct boundary values would be most stringent, if input data was generated, and the process of duplication generated.

We could also see that if some equivalence class was already exposing a problem, the test stopped working through other data in the same class. In particular, we have in table 1 below used “HV” as an abbreviation of High Value, which specifically in this case have been 999999999999999. Table 6.1 does not represent the entire set of input values; it only shows some representative examples. The boundaries value analysis selection assumes it is a value (number) scale and supersedes equivalence partitioning. For the Par-wise input, the set seems to have special meaning, thus the order and combinations could probably be varied further, but gives no additional information. Just below the pair-wise is another special set of values, where in the usage (order) of the values seems to matter. General and custom is a set of pre-set values (not available), but entering some odd values caused problems for each of these sets.

Table 6.1 Input Data related to Techniques

Allowed input	New additions	EP, **BVA** used	Magic Number used	What technique exposed failure?
Positive values	-1, 0, 1, HV	ok	HV	BVA
10000	-1, 0, 1, HV	-	Negative zero HV	Any
0, 2	-1, 1, 3, HV	-	Special set, Negative, HV	Magic Numbers
Special set of Pair-wise	-1, 1 0, 1 1, 1	-	Start on wrong number, Order	None
Special set of Pair-wise	-1, 0 4, 5, 8, 9, 5, 6, 7, 8	-	Order, negative	Negative, Zero, order
"ASCII"	space, tab, 0, empty	-	For ASCII some values could be considered	None
General Custom	Add 0 empty	-	0, empty	0, empty
0 - 4	-1,5	ok	-	BVA

6.4.2 Results from Second Attempt

For 100 input values divided into two subsystems, we had a very positive result compared to previously executed tests. With few well selected inputs according to our list, we found problems in both the software and the test application. For the first subsystem of the software, 55 inputs were handled correctly, 16 inputs are exposing a failure and 12 inputs are still in investigation, which means that either the system or the automatic test system is faulty. For the second subsystem of the software, 6 inputs were handled correctly, and 6 inputs were exposing failure and there are still 5 inputs that are under investigation. In summary, out of the 100 inputs 61 inputs are handled correctly 22 exposes a fault in the software and 17 and under investigation. In particular we want to point out that we by failure

mean, software failure, test automation behaviour (which indirectly can lead to faulty judgements on software, and also must be corrected), or any unwanted side-effect of the software under test or its test application.

The unique part of these test cases is that each test case contains several dependent input variables. This means that execution of the test case will show a dependency among the test cases based on the input being tested from a system level. The test cases and corresponding input values would have been constructed differently if these input values were executed on a code level in isolation. We also tried to evaluate how well the application of some of the most used input TDT, boundary value analysis, would find faults. We were also interested in evoking failure cases by providing negative input and select good inputs that we call “magic numbers”, e.g. 0. The major difference with the boundary value TDT to some other techniques is that the inputs can be automatically provided given a boundary. Selecting good input that provokes the software and system is often more skill dependent, and the applicability of input selection is therefore lower. Some magic numbers can also automatically be applied, but often selection and application of them are contextual. Analysis show that 40% of the boundary value related values were faulty, and about half of the failures came from entering the “magic numbers” -1 or 0.

6.5 Discussion & Conclusion

We believe the most important part of these two studies is that we achieved the goal, that with acceptable low effort we did succeed to fail test cases that revealed problems in the test application and in the software under test.

To perform research experiments in an industrial setting is not straight forward. By deploying TDTs with real testers, it is possible to lose some of the systematic approaches necessary for a better result. The second part is that it is hard to convince industry to reveal “fresh” figures from real systems. Revealing fresh data on failure is not very easy, since data also indicate a lack of test technique application for that particular organization. Therefore, we have hidden all connection with what application and organization we are talking about, though this might have been of interest. We are well aware of that any testing research has a very strong relation to what software is selected, what faults and failures that are typical for this software, and how well tested it is at the time of investigation. What

processes that have been applied before should also be better controlled to give a more educated answer. Therefore going backwards in versions of the software might be a valuable approach, strengthening our intentions for the first experiment. We think that the unique part of this result is that all test cases were fully automated. We can easily see that this will from a management point of view look like testing is well done, meaning efficient.

After this experiment, we are bold enough to claim **automation has no relation to how effective the test case** is – we assume effectiveness is related to what input you select and the number of test cases you add. It also seems that having technique aiding in selecting input values is better than a "random" value to make the test case work.

Chapter 7. Investigations on Applicability of Test Design Techniques

7.1 Summary

During a period of three years we have observed how students groups learned and transform theoretical know-how on TDTs into practical application of test case creation. The aim with this series of investigations was to find the best approach to measure comprehensibility, applicability, and hopefully get some information on the efficiency and effectiveness of TDTs. Though the initial goal was to investigate a series of different systems, during the course of this study we realized that the know-how of TDTs has little to do with the system under test. Most has to do with how the technique is presented and thus transformed into useful and practical knowledge. By repeating the same theoretical knowledge and particularly gathering data during the practical exercises, we have gathered know-how on what to measure, how to measure, and also on how to improve the teaching of these techniques. We still feel there is a need to improve the way these techniques are taught and how they in practice are applied.

7.1.1 Context of Student Investigations

The initial motivation was to explore and learn more about TDTs – in particular their efficiency and effectiveness. In study 4 we found that it is easy to misunderstand a TDT. As a matter of fact, we knew that even if the boundary value technique seemed very straight forward, it was not an easy task to apply this technique at all levels of a specific system. By defining simple systems and then asking students to provide test cases based on different techniques for these systems, we were able to measure the success of these test cases. The research question we tried to answer was strongly rooted in our overall aim of the PhD:

- How does one best compare TDTs – what are the efficiency, effectiveness and applicability of the different techniques?
- Is comprehensibility or ease of learning of a technique related to the efficiency of the technique?
- What does really "apply" a test technique mean?
- Is comprehensibility of a technique the same as understanding the application, and thus being able to apply the technique?
- Are levels a useful concept in testing software? Do students comprehend levels?

This research was done in parallel with the research studies – so the research questions were refined and sharpened during these three years.

7.1.2 Design - Research Method

The three investigations on three groups of students participating in a Masters level course on Software Testing during three consecutive years evolved from an unstructured study to a more controlled study. One could term this as an experiment, since we can very well claim the teaching of the TDTs our "treatment", and that we are studying the effects of such treatment. Unfortunately this is a quasi-experiment [215], since it has no so called pre-knowledge test, which is a requirement for an experiment and experiments also require making randomized samples. Most software engineering studies are to its nature quasi-experimental [188] due to the difficulty of randomizing the selection process.

In the first investigation of students, there was just an instruction to construct 10 different test cases with 10 different techniques during a laboratory session. We did not require that all test cases were collected at the end of the session, potentially raising some validation issues. Collecting data and controlling time is an important part of doing a study like this. The second attempt was a much more structured investigation. We had three data collection instances, two of them during classroom settings, and then one on the formal completion of a given task at home. The timings were collected according to a template, but the test cases were still in a free format. The last set of students was given a test case template, and again a structured requirement on what test cases to do, and how to record the data. This made it possible to make the conclusions that we describe in Chapter 11, the mistakes analysis. Since we will present our result for our third

attempt, we do not include but a few of the results from the third investigation in this chapter. Further details on how the three different investigations on applicability were conducted, can be found in the detailed description below.

We use [186] Checklist 3 to make sure we fulfill the adequate criteria for this multiple-case study that could be viewed as "one unit of analysis", due to its singularity of data coming from one university course, even if participants were present from several countries, continents and sometimes companies.

Checklist for Case Studies used on our three student groups:

1. *What is the case and its units or analysis?* The main focus is the group of students participating in the course for each year (consisting of three sets or cases of students). The analysis focuses on efficiency, effectiveness and applicability of TDTs (in implicitly, how do one measure these aspects adequately?).
2. *Are clear objectives, preliminary research questions hypothesis (if any) defined in advance?* The research questions are clear: Which TDTs are easier to perform (apply)? Which is faster? Which techniques found most faults? During the study we learned that we must take the overlapping use of techniques into account, the order of performance was important, the clarified reporting of how the test case was conducted was crucial to get a measurable result. Also, additional research questions were added among the three student groups as we refined them each year.
3. *Is the theoretical basis – relation to exiting literature or other cases – defined?* We are aware of the extensive source of information and literature in this field. There are similar cases defined, but most often studying industrial aspects. The most similar type of case is Murnane's [157][158][159] work, where she instead experimented with defining the techniques, and comparing few techniques.
4. *Are the authors' intentions with the research made clear?* Yes, it was advertised among the student for each group.
5. *Is the case adequately defined (size, domain, process, subjects….)?* There are limits in the size of the student investigations, which is a non-random convenience sample. We used two different systems for the first student year, but in the next two consecutive student groups, we used the same system to be able to compare them with the results gained in Chapter 8. The process was clearly defined and limited to a number of

test cases during a limited time. The strong voluntary aspect for the students made especially the first attempt results dubious, due to lack of completion in the data collection.

6. *Is a cause effect relation under study? If yes, is it possible to distinguish the cause from other factors using the proposed design?* We attempt to observe the TDT (cause) and its effect in the test case construction and design. During the three years, it is clearer when people are really using and following a technique – or not, thus, there might still be other factors that could be contributing to fuzzing this cause-effect relationship.
7. *Does the design involve data from multiple sources (data triangulation), using multiple methods (method triangulation)?* Yes, data are collected from several sources, but also several student groups or years. The methods of collection have been refined, introducing test case templates, changing approach slightly. Though, in this case, we cannot claim that there is a method triangulation.
8. *Is there a rationale behind the selection of subjects, roles, artifacts, viewpoints, etc.?* The rationale behind the selection is that observing novices learning the different techniques should be a good way to establish how easy it is to learn and comprehend these techniques. The selection of the domain and techniques was to get comparable results with industry.
9. *Is the specified case relevant to validity address the research questions (construct validity)?* We do believe that the case is relevant, though, the judgment of test cases fulfilling the TDTs must be further improved.
10. *Is the integrity of individuals/organizations taken into account?)* Yes, the students' integrity is protected, and results are aggregated in any external review. During the collection of data, due to its "intermediate" evaluation nature, the students were known to the teachers.

7.1.3 Validity Threats

In this experiment the validity of the result can be questioned on several levels. The data collected was of varying quality, where the data from the first student group was the poorest, mainly due to the lack of requirement on how the data should be defined, and what information should be provided. A lot of results that came out of this study are incomplete. It is

clear that the students often fabricate information, because they have not understood or for other reasons.

Instructions were not followed either, and data was skewed based on the misunderstandings. It is clear that data can be measured but does only communicate limited information if it is not complete in many aspects. Students are not ideal candidates for research in many cases, mostly because they are under training, and also, even if we want to measure applicability on the system – it is easy that it becomes a measurement of the comprehension, and indirectly – on how well the information was taught. For all tasks that are outside the scrutiny of a controlled environment, it is not possible to tell how much people influence each other, something which would impact the results, and can be considered a major threat to the validity. During these three attempts to collect data from the students, we learned how to better design a set-up that could at least contain some interesting data. We improved each year and student group, in our ability to measure just that what we intended to measure. We attempted to measure the applicability, but learned that we were more measuring the efficiency in comprehending and applying some test cases, and not really targeting the actual technique. The data was selective and the quality was very varied in the initial two investigations, but the more successful results were a result of the last student group.

7.1.4 Contribution

The main contribution of these three attempts was a series of value-statements on problems that exists when applying TDTs. We had some successes in measuring the comprehension in applying TDTs, which indirectly means understanding some aspects of teaching the techniques. We got some indication of what techniques seems to be simple to conduct and which seems to be harder, and also what seems to be the reasons for this difference. Secondly, these three investigations resulted in a changed view of possibilities to define what integration test means, when a system already exists. In particular does integration test seem meaningless, when the system is small in size (and only possesses one clear module or user interface) and is already integrated as one system. There is clearly a comprehension related scheme for working on different levels and thus utilizing different techniques. This is reflected in the resulting test cases.

7.2 Introduction

Teaching software TDTs should be relative simple. There exists several books, and it is a "mature" area from a research point of view, with a large number of papers published for over 30 years in the area. Additionally, the knowledge of software of an average student should be much higher than years ago – since software is everywhere. But – the fact is, TDTs are by no means a well-explored area. The plethora of redundant names, and the tendency to name TDTs not separated from the system, but more to the context of the system – makes the area additionally difficult to sort out. Unfortunately there seems to be a widespread notion in the field of software testing – that you must not know how to program or code software, when you are testing it. Our belief remains firm on this point, if you do not understand how software faults look like, and not how they propagate and are visible in a system, your know-how of test will be limited. Secondly, there is a similar misunderstanding about what test or testing is. Many confuse testing with just running and executing the system. They forget that they need a verdict and ways to determine if the system passed or failed. "Stumbling" across faults thus only pointing out when the system crashes is not sufficient, and many students do not comprehend correct behavior in a specific context. It turns out that we could confirm (even with some questionable validity) that these thoughts seem to exist within industries too.

7.2.1 Details on Research Design

This study is actually a set up based on three years, and three different set of students. The background is theoretical teaching in a normal classroom approach, including teaching TDTs, testing, test process, and test standards.

The first student group was asked to do 10 test cases each in a laboratory session. There were two different simple open source systems to be used. One was a planning system for a student (a schedule), and one was a translator with a dictionary, from one to another language. The systems were written in Java and C, respectively. The student selected the language and system they knew best from previous studies. There were no requirement (at this stage) on the students regarding how to present the result, and there were no requirement that all test cases should be completed. Due to the more "be present" and "volunteer" aspect of this

exercise, some students either did not submit data, or submitted insufficient data.

After repeated reminders, some of the student submitted some test cases for scrutiny. This only showed some very initial, qualitative results, but not enough to be treated statistically. Lessons learned from this group were that it was not sufficient to draw any conclusions about specific techniques, and that the test cases were very briefly described or defined.

The second student group, one year after, was given two clearly separated exercises. The first exercise was creating 15 test cases in five TDTs at "three" levels: Unit, Integration and System on the same systems as the year before. The five techniques were:

- Random
- Normal case
- Faulty (negative) case
- Equivalence Partitioning
- Boundary Value Analysis.

The second exercise was to create another 15 test cases based on three test cases in each TDT, where the techniques were:

- State-transition
- Fault Injection (or a single mutation)
- Exploratory (any random approach)
- Performance test
- Any other (free of choice) technique.

The third year's student group was given a test case template (see appendix 1) to fill in for each test case. This test template also included a test record – where the students were to fill in the result of executing their test case. This was also as a measure to ensure that the test case was executable, and that failures were properly logged. It also gave an opportunity to reflect the verdict of the test case to the actual outcome of the test case in a better manner. The system here was the same as used in the industrial study in Chapter 8: Buddi[32], to enable an easier comparison between the students and the industrial testers. In fact this third year study was conducted after the industrial study was performed.

A clear set of TDTs was used, both individually and in combination, and the teaching was much more targeted on these TDTs. At this third year student group, no "levels" of test was used or discussed.

The ten test cases were in fact only 8 techniques, where techniques were: Positive Test, Two test cases for Negative Test, Equivalence Partitioning, Boundary Value Test, State-transition Test (with model), State Transition Test (with table) and at least three transitions, Permutation test, Combination of State-transition test and Equivalence Partitioning. The way the practice and data collection was performed differed in the three studies.

7.2.2 Results of First Student Group

The first student group was relative small, and also had little direction. The remaining impression is that the students, who were given one extra lesson on state-transition test (model based test), did find it easiest to create test cases based on this TDT. All were given proper state transition models. Also, the normal/positive test cases were dominating. In fact, it seems like very few other techniques, than these two were visible in the resulting test cases. It was not only that the test cases made in the state transition model were dominating in the delivery they were also the most well made test cases. Many of the other test cases were poor in construction, and very hard to tell what they were really executing. In fact, it was hard to distinguish any other TDTs, apart from positive testing and state-transition testing. Here we again saw evidence that boundary value testing is often mixed with "outside the boundary" by several students. Since none of the results were significant, we present it as qualitative lessons learned.

7.2.3 Results of the Second Student Group

In the second year student group, the students were required to name the level (unit, integration and system) where to perform the test cases. For these two exercises, a data collection template was given, where information was to be gathered (see appendix x and y). For the first exercise, data collection was performed immediately after the first introductory lecture – and repeated in full for the entire sheet as a mandatory part of the course.

During the class, we also took notes on all noteworthy actions during the exercise. In the study, there were 28 participants, with an average age 25 with average experience of 2 years of programming. Most people got started immediately and played with execution of the system, trying to understand the purpose of the system. A few individuals had not started after 30 minutes, due to series of problems of understanding the exercise or problems with the hardware/software instructions for download. The range of finishing the exercise was about 0-8 test cases (of 15) during the 1 hour and 30 minutes attempt, where it was often the case that not all fields were filled in.

Since two collections of data were done, the immediate collection for the students (initially during the class-held introduction) was that 16% of the total number of test cases was attempted. For the second collection still not 100% of the test cases were completed.

The initial goal was to measure the tests cases at each level, thus each of the 5 types of techniques were to be repeated on three "levels" of the system; the code level, the integration level and the system level. For the initial collection, the average is 2.8 test cases per student and median is 2.5 test cases per student out of 15, where 5 integration test cases should not be possible for this small system. None of the systems had a clear integration level, and the request did not make any sense for the students. About 15 of the 28 students seem to grasp the concept of system versus code. A few students with code-skills did occasionally define how they viewed the integration and constructed test cases according to their own assumptions.

The students also had an escape route to take, that they were not able to apply the test case on that level if it was clearly stated that they have "given up". This instruction was not understood by all, thus many students plainly ignored handing in test cases, thus did not follow the instruction. 18 students got stuck or skipped attempting to create a test case for integration level, whereas only one clearly marked it as a "given up", which was requested as an answer and 3 pondered clearly on how to do this for 20 minutes. 10 students out of the 28 attempted to construct a test case for integration. For a few (less than five) of these students – numbers were clearly constructed, or the level aspect was ignored and just a "random" test case created. This gave us important indication that talking about levels comes much more naturally in a large organizationally divided system, where the "test object" might be the integration level – but the user sees this as the "system" they are testing. Only 3 of 28 students could see

through that the estimated number of possible test cases for this (and any system) is plenty or almost endless, whereas the majority of students wrote down a very limited number on how many test cases it was possible to apply.

Even in the short time (during the initial class-held introduction) the students found 21 faults. But – many were influenced by others – so the faults found were not unique. One could clearly show that the second lab and techniques like negative testing, and performance were in particularly difficult for this group, and had low completion rates. As a technique, Fault injection was also often misunderstood, but since the system was of such poor quality- e.g., it was possible to store a scheduled class with the start-time after the end-time, and it would be saved and shown in the system – which then worked as an example for (all) the students.

Most students did in fact random executions (exploratory test) and stumbled across problems. It can be seen that test cases identifying faults were favored by the students, rather than following a systematic approach. Furthermore, it was also common that the students found the same faults and there is again evidence that the test cases were very similar – since no restrictions were imposed on the interactions among students.

One major problem with this investigation, is that almost all test cases failed to show that a specific technique was used, but there are frequent indications that "some" part of the TDT was used. For example, performance test did measure some aspect of time, and boundary value testing at least attempted to identify an "out of bounds" value. This led us to seriously question: How does one measure or "prove" that a specific technique is used? The fuzziness in this aspect really originated from the problem that almost all test cases failed to be completely and adequately described. A few students defined good test case tables to describe the test case. Most used a very brief execution path, but lacked any form of verdict (except where test cases clearly indicated a failure), starting position or enough execution information to be able to repeat the test case. The best described test cases were those that triggered a perceived (to the user/student) a failure. Therefore it is interesting from an industrial point of view that the students were asked to record the failures with adequate information. In principle only 3 students gave sufficient information to be able to file a defect report.

An interesting side-effect of this investigation was that it was very easy to clearly identify people with skills to look at the code. It seems like 8 students fall into this category. All of these students reported high numbers in understanding. It might not be enough evidence to strongly conclude this, but it corroborates with many other indications, that code level understanding of software is a must to perform adequate test cases. To be able to tackle the software and test of systems, many abstraction levels of the system must be understood. It seemed that students could express themselves with more confidence and detail if they also understood and could perform at the (same) code level. One could identify that fault injection was useful on both code and as "faulty" input on the system level. Most test cases were through the systems interface/GUI – thus on system level.

The range of "creativity" of test cases (we asked for unique test cases), shows that the use of a technique for the test case creation had a very "free" or fuzzy interpretation (or contributing factor) in creating the test case. As we will show later, this is the first sign of the clear overlap that exists among definitions of TDTs.

There were five persons (of 28) who understood the state-transition technique, and could construct a "model" that they provided. These students had better background and grasp of testing as compared to the other students. They had also attended classes in modeling and testing, and were thus better trained for this task already from the start. This showed in the overall result.

Times collected for this group, have very low accuracy. Since they are a reflection of the system, the exercise, and in this case the students either trying to look good (low numbers), or "punishing" the exercise by giving it very long times (compared to the researchers notes). Also, the timings can only be taken into account if the result is true – meaning, that one can show that a test case actually is a test case in the technique and on the right level. Otherwise one does not measure the technique, but are judging the testers "innovation" of a test case. The result on this behalf in this study was so poor, that we do not find it worthwhile to ascribe the time to a specific technique. This could be viewed as an example of an "uncoordinated" case study. Nevertheless the data was collected, and can serve as a pilot guidance, with reflections and lessons learned made upon them.

- Time to understand the technique and where to apply it ranges from 1-15 minutes in most cases, average around 5, but the first test case sticks out often having 10-25 minutes as a starting point.
- The time "where to apply" the test case in the system for a technique seems in this study to be very low number (between zero seconds to a few minutes, as a rule, less than 5 minutes) and probably goes together with the first measurement (understanding the technique at all).
- Time to construct the test case was for almost all students a systematic number lower than the time to understand the test case. This means below 5 minutes, with only very few exceptions. It shows that once the problem (place) is understood, the actual test case creation takes little time (when the system is at hand).

We also asked the students about their ability to grade the difficulty to identify the location where to apply the test case – or the similar rating asked about to grade the difficulty to create a test case – could not be concluded to follow a specific pattern. These columns seemed superfluous. This concludes us to advice against measuring a qualitative variable like "Perceived difficulty", but instead measure the actual success of creating a unique and valid test case.

With this discussion on integration test and the use of levels, we can highlight the following conclusions as emerging from the second investigation of the student group:

- There is no pattern that indicates level in the test cases.
- The lack of integration view or focus when a small system is created, and also when the system already exists – one tend to "use" the system rather than creating test cases.
- Understanding and or following instructions poses difficulties.
- Information in a test case is often communicated at a too high level, and is not detailed enough.
- There is strong evidence of different type of "overlaps" of the TDTs. The same test case can be used for different TDTs.

7.2.4 Results of the Third Student Group

In the third investigation, we focused on remedies for problems found in the previous two student groups. In particularly we wanted to create an investigation, where the data collection was in a similar form that could make it easier to judge and determine if the test case was performed correctly.

The third study had as already stated a description of what techniques to use, a Test Case Template, and also an adjoining Test Record, as seen in Appendix 1. The results gave us measurable data to be used to verify the success of the technique (applicability), and measurements on timing, where we so far have ignored the data for the timing – since they are qualitatively recorded. We suggest for the future that this data is automatically recorded, for achieving a much better accuracy of the collected data, and not manipulated by the students. Furthermore, a good way to really measure test case creation would be a full recording of how people work with the system, which is probably costly, but would be a better "proof". To our astonishment, and also taken into account that the detail and lectures for the area had improved, our results were still dissatisfactory. Instead of getting a several hundreds of well-written test cases, we ended up with several hundreds of test cases that were flawed in many different ways. We still had problem with the level of detail, ambiguity of verdicts, lack of starting positions and too little detail in the step-by step instructions, which at one level can be understood. These are students in the process of learning these matters and we collected an intermediate result from them. Nevertheless, the quality of the data was much improved, by sheer numbers. We could see a great trend of which techniques were perceived easy and furthermore, see what was mostly tested. Yet, a series of problems still remain, where we now instead focused on another aspect of the test case creation process: What mistakes are most common. This means that the main result in the third student investigation instead resulted in a study of mistakes on test case construction that is described in Chapter 11 (Study 9) [70].

The result of this third attempt was aimed at 10 test cases per student, but many students failed to hand in test cases for all techniques, simply used one technique all over, or skipped techniques that looked difficult. Therefore our statistical result indicates what was reasonable performed and attempted. It would be possible to collect the data for the timing to apply these test cases, but currently we believe we must have a different

way of measuring this – since students will often not write the accurate number. This is probably both conscious and unconscious skewing. There was a deliberate focus that the students should define repeatable test cases that were sufficiently described – and the usage of the template we hoped assured that aspect. As the result showed, there were many fallacies in the result, and these test cases would simply be inadequate for use in the industry. This observation made us focus our efforts on finding out what is the hindrance for producing adequate test cases. Even if performance was overall much better, it became clear that the test know-how and skills transfer is a real issue.

7.3 Discussion on Measuring Applicability

The above lessons learned made us focus on the important notion of "measuring the right aspect" of applicability of TDTs. It seemed like we instead measured how easy the system was to grasp, and to execute it randomly, than the ability to create possible test cases (this is with a few exceptions of some of the students). More focus must be placed on the difference between constructing a "random" test case and constructing a test case that actually is a result of deploying a TDT. This means that more focus must be on following an accurate "rule" based on a clear definition. It must also be clear that when techniques overlap – it is hard to show evidence of that they are producing a result that is different.

We can also see supporting evidence that some techniques seem better understood than others. In fact in this group, as much as 15 persons (actually divided in 3 groups of the first student investigation) decided to work further on the fault-injection technique, and out of these 10 (2 groups) showed a significant depth of understanding the technique and its purpose. These students were more interested in looking at the source code, and thus took a second look at this exercise. It gives an indication that the best teaching approach would be to stick to one "group" or one particular technique at the time, and instead go into depth with each of these techniques to get adequate feedback.

We suggest that our future investigations should explore the competence of transforming development specifications into a test case using a TDT and should not be based on the system or user interface. Not always does a

system exist to base the test case description on and especially not for new development.

This is also based on the industry need, to have new software fast on the market (time to market), whereas the test execution time on the critical time line should be minimized. This implies that automated test executions should be created (based on the specifications) before or concurrently with the creation of the code.

These series of attempts to collect data from the students shows not only the significance of having adequate means to determine the actual outcome of the results, but also gives an indication of the cost of performing such investigations. The data should at best be collected and treated statistically on-line, but not handled on manual collections. Also, the high degree of freedom gives indications of the know-how of the problem, but also causes extra strain to define and decide limits that in the end results in validity problems. Research design is of outmost important to get scientific viable results.

We could through these different student groups see an improvement by more precisely defining the expected result formats. Our third attempt gives us important results, where one can determine if the students fulfilled the TDT or not, thus start discussing the applicability on a more in depth level. We have a strong indication that lack of know-how on how to systematically apply the technique is the actual hurdle for usage of some techniques, which means the technique must be well understood to be able to apply the technique to different type of systems and contexts.

Chapter 8. Applicability of TDT in Industry

8.1 Summary

In a series of studies we measure the know-how of TDTs of a variety of experienced developers and testers from industry. Our result shows an on average knowledge of 4 TDTs, and an ability to apply two (possible three) of these. The most known TDTs are Positive (requirement) testing, Negative Testing, Boundary Value Analysis and Stress Testing. Only very few persons have an in-depth understanding of the TDTs. These people seem to have a general interest in testing and an extensive education on the subject. Only one of the persons could see through the overlapping set-up of the TDTs. This indicates there is a great potential to improve and utilize TDTs in the industry. Thus with education and understanding the test case quality would improve dramatically (and as a consequence, the coverage and the fault finding ability – thus both effectiveness and efficiency).

8.1.1 Context of this Case Study

The motivation for this study is to determine the know-how of TDTs. One of the research questions we wanted to get more information on was, that even if experienced testers and developers that work with constructing test cases for years, would know and use a series of techniques, they maybe do not know the name of the techniques they used. A part of the study was also to verify that "knowing" really meant that these experienced people could describe test cases for a simple system – even if not familiar with the system itself. To be able to "sell" such a study to industry, a part of the study included teaching some fundamentals about the technique. The intention was that we would get a series of test cases of varying quality – and thus be able to determine the depth of know-how, how easy it was to create these test cases, and as usual, with a very strict time limit create a series of test cases. The intention was in addition to do a triangulation on data collected in Chapter 7, to reveal the number of people that could

rapidly grasp the study, thus within the hour create 9 test cases. The "solution" would have been to create one test case (execution path), but vary the input depending on the different techniques. The actual result was that only one person succeeded in doing this.

8.1.2 Design – Research Method

The design of this quasi-experiment was a survey (see Appendix 3) where the first part collected back-ground information, by checking the current know-how of TDTs. Then a lecture took place, where most techniques were explained, with particular focus on the techniques used in the subsequent applicability study. The next step was then to measure the improvement in knowledge of the techniques – where the techniques not mentioned are used as reference point. This was then directly followed by the actual creation of test cases for a simple system. The idea was to measure that the people understood the techniques and could apply them correctly. The result was collected and statistically handled. The survey was first used as a pilot, leading to adjustments in timing and removal of some faults in the questionnaire. Then the study/survey was repeated for several groups in different organizations and companies. The groups were comprised of 5 to 10 persons to give adequate time to respond to questions and to set up the system and techniques.

The main research questions we wanted to get answered in this study was to get confirmation, that people in industry know TDTs, (even if they do not know the actual name of them) and use these technique regularly. This two-part question was clearly answered in this study, but not the way we expected. Further, we use the checklist in [186] (Table 3) describing case studies, to check that all aspects were answered. In addition to what is described in the sections below (where more details on set-up etc is revealed) the following is a short summary of the answers these questions.

1. *What is the case and its units or analysis?* As specified in 8.2.
2. *Are clear objectives, preliminary research questions hypothesis (if any) defined in advance?* Yes, as specified in 8.2.1
3. *Is the theoretical basis – relation to exiting literature or other cases – defined?* No, no related work (other than what is stated in the previous study (Chapter 7) is added.

4. *Are the authors' intentions with the research made clear?* Yes, a preliminary information sheet was sent to all participants upon invitation.
5. *Is the case adequately defined (size, domain, process, subjects….)?* The size is adequate, 47 participants, but the selection is only from two companies, with at different geographical and organizational selections (thus different software and systems). The process was refined after the pilot, and small justification was made to add more (redundant) names of the TDTs.
6. *Is a cause effect relation under study? If yes, is it possible to distinguish the cause from other factors using the proposed design?* Yes, as described in Chapter 7, there is a cause (the technique) and the effect (the actual applicability of them that is compared.
7. *Does the design involve data from multiple sources (data triangulation), using multiple methods (method triangulation)?* Multiple data collections (sets of same study repeated), but same method in all but one occasion.
8. *Is there a rationale behind the selection of subjects, roles, artifacts, viewpoints, etc.?* We wanted particularly to select experienced testers and developers, who in their daily work write test cases as a part of many tasks.
9. *Is the specified case relevant to validity addressing the research questions (construct validity)?* Yes, we believe so.
10. *Is the integrity of individuals/organizations taken into account?)* Yes, all individuals and organizations were coded with a unique ID, kept separately during the process.

8.1.3 Validity Threats

The main validity threat is the limited number of industries that participated, and that all the studies were performed in Sweden. The participants came from a wide variety of nationalities and countries, partly removing a cultural bias. The majority of participants were from the telecom industry, but a series of different products and levels of products (from middleware to applications) were involved. Secondly, the selection was not researcher biased, since the selection was made by managers, but there is a threat for any industry study that the people who participate in these types of studies/education are seldom key personnel, even if we noted

that a few key personal participated. Thus the representativeness of the groups could be discussed. The variation on experience ranges from 1 to 30 years, but with an average of approximate 10 years of experience and 8 years median. In the study, both developers and testers, with a few managers, consultants and other type of leaders and roles were participating. Since many industrial people felt comfortable to discuss, the participants in each group might have been influencing each other. Since time-limits were used and the order of the test case techniques were scrambled we could obtain information on if it was the time limits or know-how that was restricting the test case construction. There is data triangulation, though the data was collected at different stages. There is no method triangulation; the only change is the order of which test cases appear on the data sheet.

One can claim threats to construction validity using humans that has a tendency to over-evaluate their result, and answer what is expected, see Yin [217]. One could also discuss the fairness in asking people to construct test cases on a novel system under stress in a very limited time-condition. Are we really measuring the people's ability to use TDTs, or are we using their ability to understand "any general instruction" and then apply it? The nuances between these two statements differ as their interpretation. In this case (as in most involving human subjects), we should be very careful drawing too many conclusions.

The study can be repeated, whereas the teaching part of the study might have different qualities (which could be considered a method variation) at the different occasions. This might influence the result. The collection of data could contain some judgment that is researcher biased depending on if the result (the test case construction) can be claimed a valid example of the technique or not. This influences the result. To minimize the internal validity threats, the category of "hesitant" was introduced, to make it clearer where the data categorization might contain problems in interpreting the result.

8.1.4 Contribution

The result gives interesting and strong indications in spite of these validity threats. 47 participants were contributing to the data in the study, but the first group of 5 (pilot), only one submitted data on the test cases much after

the survey and "treatment" i.e. teaching, which changed the experiment set up, to collect all data on the spot. The main contribution is that we could establish that there is not a lack of knowing the names (theory) of the technique, but a lack of know-how on how to actually apply the technique. It seems a strong indication, that the techniques that people said they know prior to the study are what they really know and can apply. In particular, it shows that the Boundary Value Analysis and Equivalence Partitioning are techniques people think they know, but in general do not really know in practice. The same seems to be true for Magic Values and Permutation, but we could not conclude this finding, since there were too few attempts to perform this technique. The most common confusion seems to be to test "outside" the boundary, which is a negative technique, instead of performing a proper Boundary Value Analysis series of tests. This study shows that more knowledge is needed in the area of TDTs. It seems also very clear that all that attempted normal test cases could perform them correctly – making it the most simple and obvious TDT.

8.2 Introduction

The main goal of this study is to learn more about the know-how and use of TDTs in industry. The overall goal was to better define what applicability means:

- How easy to understand is a technique?
- How easy/difficult is it to apply the technique on different types of systems?
- How much "human creation" does the technique require?

In addition to this goal we are also interested in the efficiency and applicability of the TDTs, or in other words how "good" the technique is (repeatable, automatable) and thus useful for industry.

TDTs, or TDTs are often taught at courses, or defined in syllabus, test course materials and "standards". It is easy to assume that these are well known. But we did not know which techniques were the best, and which were used and not. Looking at test cases and the result in industry really made us question the spread of these techniques. Is this a fair result, given the quality of the test cases using these techniques? If the TDTs are know and taught why they are not really visible in the resulting test cases? This

inspired us to collect data as precise as possible, and trying to answer some fundamental questions that the industry need to provide including

- The time it takes to perform a certain tasks
- How "well" people claim they understand a TDT compared to their actual success in applying the technique
- Judgments on difficulties of TDTs
- Any other valuable information that could be obtained regarding the industrial understanding and use of TDTs and their resulting test cases.

Since we already had done an initial study (Chapter 6), and started on the two student groups (Chapter 7), we had some initial data, on which we could make further data triangulation. We wanted to collect more data and cross-reference results to improve on the validity. To make industry interested and make the study more easy and fun to participate, in hope not only to attract people that had time or were commanded by their management, we suggested to hold a little mini-course on TDTs, as part of our study.

There were small variations in the survey. E.g. some questions were unique for designers/software developers and for testers. Also, some corrections of faults and additions of techniques changed the survey between the first three groups. Then the survey was kept similar. There may however still be slight differences in how the TDTs were described in the different lectures. The Pilot group of 5 persons were not required to hand in the test cases and did not use the sample system, but attempted the techniques on their own system. This resulted in that only one person handed in the data, which made it difficult to compare the results. The result was applied correctly was much harder to determine for the researcher in a foreign system, containing much specialist knowledge. Therefore we omitted this result as an outlier. The other collected data except the applicability test could be used for all five persons in the pilot study.

8.2.1 Research Questions

Our main research questions are:

1. How many TDTs are known and used in industry?
2. Do testers and developers know how to apply the techniques correctly?

3. How many understand the overlap of most TDTs? (Thus, are able to produce all nine test cases within the time limit?)
4. Are there preferences of which techniques are used (and used correctly) and which are the techniques not commonly approached?
5. Are there patterns on what is selected to test in the system?

8.2.2 Study Set-up and Selection of Targets

The study was made in groups of maximum 10 persons. There was first a pilot group consisting of five developers – attempting to collect test cases. This did only result in collection of some administrative data, but the actual test cases were never conducted, which led to poor validity. This group also insisted to deploy the techniques on their own software. The relative inexperience of teaching the techniques suitable for this type of software resulted in a hesitant result, where only one person finally completed the test case result. This could not be used in itself, since no comparison existed. Determining if the test cases were in fact obtained by using the technique can be discussed. After this initial try, we enforced a much stricter policy around the study, checking that people could stay and fill in the result, and having a stricter time policy. We then invited people to participate in the study across companies, but only two companies responded. In hindsight, this should have been done using a different selection method, but it gave us a rather big selection of organizations within one company.

Other facts around the study are the following:

- Timeframe approximate 3 hours, including 30 minutes to finish the first part of the questionnaire. A 1h 30 minutes mini-course, and then 1 hour to write the test cases for the considered system, including a short break.
- All data collected used standard ethical rules – meaning all identities are hidden and no individual results are presented.
- The participants are all getting access to the final result (this thesis).

The agenda included the following steps.

- Introduction, aspects of the study, motivation and goal

- Collecting data, background information and existing knowledge of TDTs
- Walk-through of some of the techniques – to enable the participants to match a technique and a name of a technique
- Introduction to the system for a few minutes
- Creating test cases for the system and recording them
- Short summary and discussion/questions and collecting the results.

The results were then aggregated by the researcher and treated statistically. Some of the results are presented for the first time in this thesis.

8.2.3 System Under Test

In this pilot study we did try to make the set-up as described above, but let the developers using their own software and hand in the completed test cases. Only one developer concluded the task at all, which is not sufficient and shows the difficulties in obtaining industrial results, and enough results. The only data we managed to collect was the "background" and before and after assumed know-how of the TDTs. Since we lack any "proof" in the form of test cases demonstrating the techniques, the result was rather disappointing from a research point of view.

Therefore we instead turned our focus in the actual study to use the system Buddi [32]– which is small system that is easy to understand and has a simple GUI that could be learned in a very short time. Also – moving away from familiar systems, would more test the ability to perform the TDT, than the actual know-how of the system under test, and this would contribute to a better measurement of the TDTs. Buddi provides a simple budget management for personal use, where you can register accounts, make a budget, and register spending to get an overview of your own financials. It contains several faults propagating to failures in the user interface. It is an entirely string-based handling system, assuming you want to solve your own problems. We used an earlier version of the system deliberately, to also have more flaws visible for the user, since to our experience, this is a motivating factor. Also, these faults can serve as a check-point that the system is tested thoroughly.

The lessons learned from the second study (see Chapter 7) were that the test cases were only to be defined on the system level (on the GUI-level) of the system. See also further discussion on level in Chapter 13.1.2.

8.3 Results

This quasi-experiment tried to collect background information before the treatment. Since the data was collected over a longer time period, we did add control questions, meaning – our resulting data are not fully completed for all questions and on all techniques. In particular, we attempted to add techniques with similar names, attempting to check if one name or variant was more known than another. Unfortunately it looks like most of the people see every entry as a unique TDT, and only one person asked specifically about these variants. Out of the group we could conclude that only one person really carried the full skill set of the TDTs and could "see through" the entire experiment, in the sense that the same techniques had the same ratings and also that the applicability of the different techniques was really doing variations of the same TDT input, but with small differences in input and way to state the test case.

8.3.1 Background Information on the Selected Group

Experience of the testers and developers varied between 6 months to 30 years experience. 22 persons had testing as their main work task and 25 had development as their main work task. Consultants and design managers are divided into the group based on their dominating number of years. Average coding experience in the entire group is 8 years, and median was 5 years, but among the developers the average is 11 years experience. The average testing experience among all was only 4 years, with a median of 1.5 years. The combined average of experience was 10 years. One explanation for the difference in experience between testers and developers is to compare the age distribution with the year's experience, and it is clear that many newly recruited start as testers. The median age-bracket is 31-35 years of age for testers, but is 36-40 for the developers'.

One other possible explanation is that one can view testing as an occupation that you start your career with, requiring comparable little experience compared to, e.g., development, and people are staying much shorter time in the profession, which of course also would influence the result. The age distribution is shown in *Figure 8.1*.

Almost 81% had some form of university degree, but only as few as 3 persons had an ISTQB Foundation certificate in test. The nationalities of the participants varied, but a majority had a Swedish background. One can also note that very few had any education in test through their university training. Also very few had a dedicated software education; the most common education was a general Master of Science.

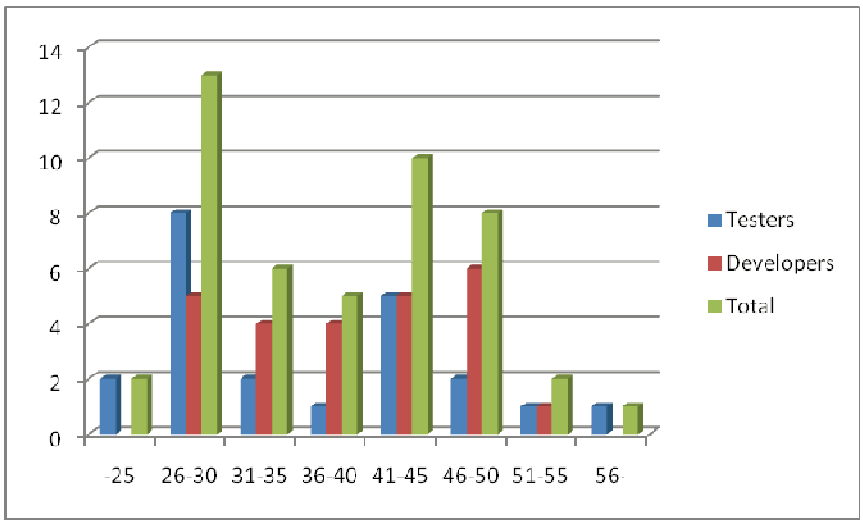


Figure 8.1 Age distribution of the sample
x-axis: age and y-axis: years of experience

There were more people modifying existing test cases than doing completely new test cases. Average frequency for making new test cases was a few times a year. Averages for changing test cases were monthly to bi-weekly. One developer claimed (s)he did write test cases before code, 9 claimed it happened sometimes, and 10 claimed it never happened. Of the 25 developers answering the question if they ever did delete code that was not tested, all 25 responded “Never”.

8.3.2 Results on the Know-how, Usage and Applicability

In the pre-test of the experiment, we measured the prior knowledge and understanding of the TDTs. The most familiar TDTs on the scale 1-5 where 1= "know-well", and 5= "never heard of" had the following distribution for the TDTs. None received the "top" number 1 (know well). The following techniques were scored as the most known TDTs (2): Requirement, Functional, Use-case, Negative TDTs and Boundary Value Analysis Technique. The second group had the average value of 3, includes the following TDTs: Exploratory, Normal (positive), Stress, State Transition, Random Input and Random Test, Fault Injection and Statement Coverage. The rest of the techniques scored 4, except the Fast Anti Random Test that scored 5.

This helps us answer the first research question: *How many TDTs are known and used in industry?* The answer according to our study is 5 + 8 = 12 techniques. This particular answer might have several problems. First, not all TDTs are listed. Also, it is very simple to claim "you heard" of the TDT, but as we will see, this must be compared with the actual success in applying the test case technique. We could conclude that it was much easier to claim you know about "a name" of a technique, than actually being able to use it.

In *Table 8.1* we have combined the most known techniques with the order in which people claim they know and are actually using these techniques. We can call this set the most familiar names on TDTs. It is interesting that Statement coverage ranks in the second group, but is not much used. Not surprising, functional test is a well known view of how testing is supposed to work, immediately followed by the similar TDT, requirement test. Interesting is that people are familiar and are using negative test approaches, but Normal TDTs are not most known, but used by those who knows it. For this technique we think the name was more confusing than the technique, and in this case, normal or positive test confirms our initial research question. After negative techniques, use case testing and boundary value analysis are well known and used. The variance is in this case the spread or differences in answers, where the most difference is found in normal test and followed by exploratory test, and then boundary value analysis.

Since in principle functional test, normal or positive test and requirement test are the same test approach – we can already here see that people do not infer this logic. Similarly, Model-based Testing and State-transition testing are interchangeable, and should receive the same score. The study is not complete in this aspect. There are more TDT names in regular use – both synonyms and variants. The complete list that we investigated in this study can be found in the Questionnaire in Appendix 3.

Table 8.1 The most familiar names of TDTs

Most known	Test Design Technique	% TDTs Known and Used
Yes	Functional test	72%
Yes	Requirement test	53%
Yes	Negative test	51%
	Normal test	45%
Yes	Use-case test	38%
Yes	Boundary Value Analysis test	26%
	Stress Test	19%
	Exploratory test	17%
	Random input test	13%
	Fault Injection test	11%
	Model based test	9%
	Equivalence class	6%
	State transition test	6%
	Statement Coverage	2%
	Random test	2%

To follow up on the above table, it is equally interesting to see in *Table 8.2* what the people rated as TDTs they know, but are NOT using among those TDTs scored in the top two groups. Most interesting is that almost every third participant would like to make Boundary Value analysis testing and Stress testing. Note that these figures in both Table 8.1 and 8.2 are a % related to the entire group.

The explanation given for not using these techniques was for a majority of the respondents' time limitations, but also quite a few testers responded that this had to with the system under test, and that it depended on the type of system.

Table 8.2 The most familiar TDTs NOT used

Most known	Test Design Technique	% Known and Use
Yes	Boundary Value Analysis test	28%
	Stress Test	28%
Yes	Use-case test	17%
	Statement Coverage	15%
	State transition test	15%
	Equivalence class	11%
	Model based test	9%
	Fault Injection test	9%
	Call-Pair Test	9%
	Pair-wise Test	9%
Yes	Requirement test	9%
Yes	Negative test	6%
	Combinatory	6%
	Cause effect graphing Test	6%
	Negative Test	6%
	Exploratory Test	6%

Furthermore we could measure that 75% believe that it is possible to use the technique to do more test cases of the same sort, but 55% claim their testing is input dependent and 53% of those who answered the question believe their test cases are effective.

We could see that on average, the improvement on the TDTs was increased two steps on all TDTs. This would have been a great evaluation on the "treatment" according to the participant in a normal survey, but since we deliberately did not teach all techniques, several techniques should not have changed their status at all (and served as a "control question"). This indicates either a willingness to support the teaching activity, or as we suspected much later, that an overestimation on how well the technique "is known" is actually a measurement on "How well known or familiar" the name of the TDT is, and not the how known in the interpretation of having the skill to apply the technique in reality. I think this is the mind-leap we humans tend to do. Because we have a hunch on the theory of a technique, and are familiar with a "name" (having a way to remember and identify this

technique, we assume this means we "know" it. But there is a huge gap between knowing a name of a technique and a "general theory" – and the skill to apply it is vast. And, therefore direct training in how to transform theory to practice are a necessity.

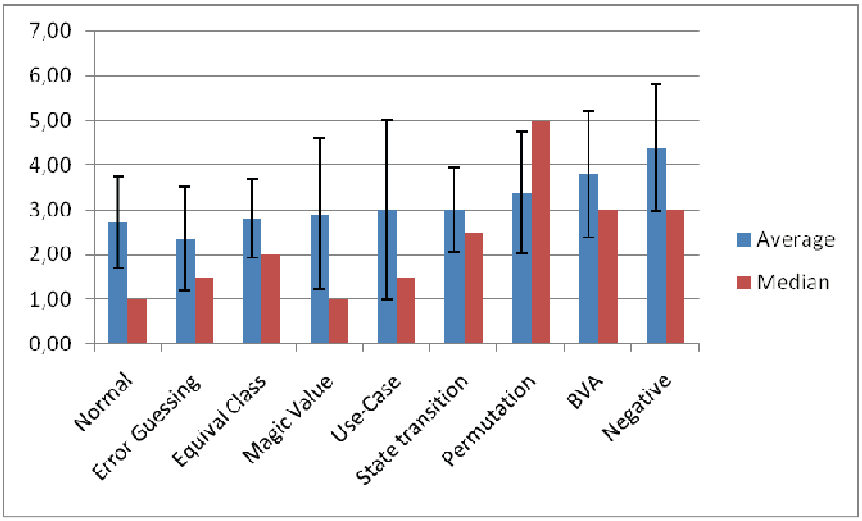


Figure 8.2 The average and median time (y-axis: minutes) to identify where to apply the test cases for each technique necessary, showing no significant difference for p=0.05

It is interesting to view the data in this study, where as a researcher, one must understand the rather difficult task of learning a totally alien test system within a few minutes (approximate 15-20 minutes is spent in each attempt). The system is not very complicated, but it becomes clear that a good introduction of the system is important. In the first few experiments we are not giving much help on the system – afraid of leading the users into limited thinking, but later we realize that a few minutes of description is enough for people to grasp the general aspects of the system. The result is that the quality of the test cases improves slightly, but the downside is that people attempt the parts of the system mentioned in the overview – which is reflecting in the results of the test cases, and what people target as test cases. Figure 8.2 presents the time it takes to find a location to apply the techniques used in the exercise after the lecture, regardless of success in the application. Since this table is not correlated with the success of how

people actually performed in applying the techniques, it must be used as a very coarse indication.

The most interesting to note is that the three highest scoring techniques were all including negative aspects addressing the input data. In addition, equivalence class should also have been in this selection, but since it is one of the most misunderstood techniques, it is not so strange that people often omit values, or in this case classes, that are more difficult to obtain. In *Figure 8.3* the average and median time to create the test case is shown, also without the actual success-rate.

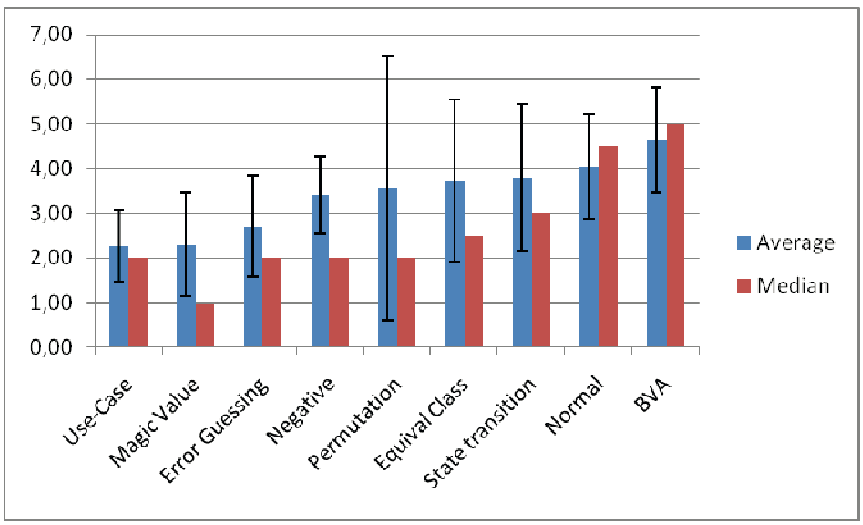


Figure 8.3 The average and median time (y-axis: minutes) to create the test cases for p=0.05

The average and median time to create test cases is significant longer for the Boundary value analysis compared to use case techniques, but there is no other significant difference shown. This is easily explained since was made that at least three values (three unique test cases) were to be written for these tests. That normal or positive TDTs takes longer time probably has to do with that this is the test case most people choose as the first (all except 3 persons). It also makes sense that state transition and equivalence class took longer time than for example use case testing. Better is if we combine these two timings and check what the total average and median time for finding somewhere to apply the TDT and creating the test case. This is done in *Figure 8.4.*

We can conclude that Error guessing is the simplest technique, since you just guess – but it is ok to "guess wrong". Magic value is equally fast and Use-case was very familiar, and is basically "any" execution path as soon as you understood the system. One could argue that the Normal TDT would be equally fast, but we suspect that it took a bit longer, since it is "the first" test case most people in the different experiments started with. Therefore we attempted to change the order in the last 4 trials of the different test cases, but people started with that test case anyway, so that did not affect the timing or result as far as we could conclude.

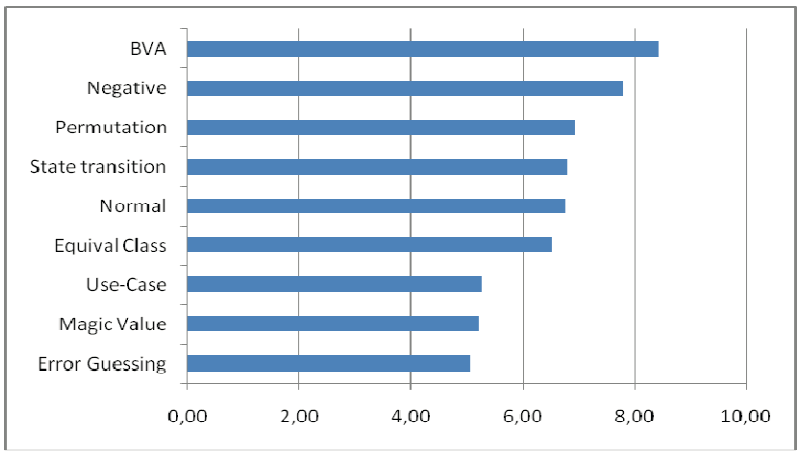


Figure 8.4 Combined average time (x-axis: minutes) and order of test cases, including both where to apply and time to create the test cases.

That the other techniques did take longer time sounds plausible, since they require more effort, analysis, more data, and more writing down in the experiment. From an industrial standpoint, the time would not make much difference, since effectiveness should be taken into account as well.

We can further conclude that this table provides a very good triangulation of data for the timing of how long the TDTs take to apply, with the timing done by different systems and testers under different circumstances in Chapter 9 and 10 for the similar techniques. There is not a huge significant difference in time to apply these techniques on average. And that the variation seems relative small. The average timings in the forthcoming studies were shorter than these average timings, probably due to more practice and training in the different techniques.

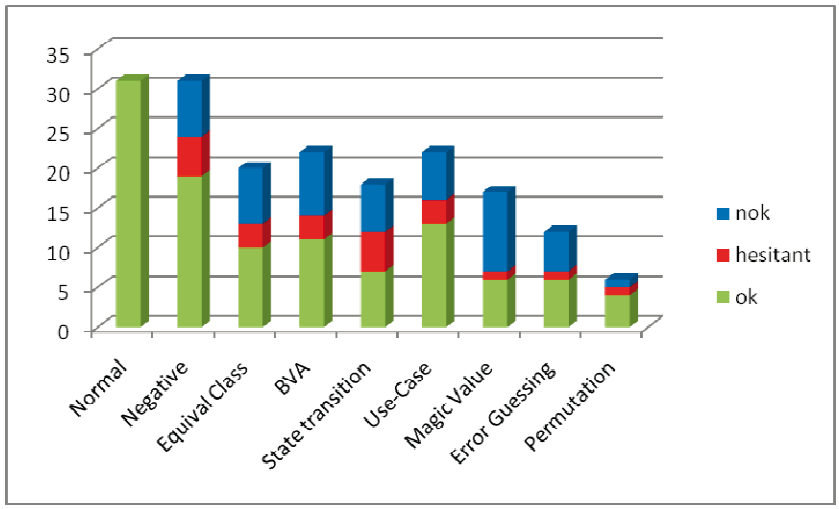


Figure 8.5 Applied knowledge of TDTs
(y-axis: Number of Respondents)

This result answers the second research question: *Do testers and developers know how to apply the technique correctly?* We can conclude for normal (positive) test technique, that the answer is clearly yes, and it seems true for the Use-Case techniques too. Especially for all normal test design this is clearly true. Then doubt enters, when not more than half of the test cases are clearly ok. One can also wonder why not more persons have attempted e.g. Magic Values, Error guessing or Permutation. Working under "pressure" in a short time frame might be one explanation. Another is that these techniques are just not well understood and applied.

The success of applying techniques gives a totally different picture, as we can see in *Figure 8.5*. The order, in which the techniques are presented in, are also the order in which most participants were presented to the TDTs. We can clearly say that all Normal test cases attempted are actually fulfilled. It also shows that the obvious, intuitive test case is simple and executes without finding any fault. We believe that the number of attempts at the end partly can be explained by the lack of time, but one can also claim that it has to do with the lack of feeling secure with the techniques, since in the later experiments we had the order changed and swapped around. Though, most people actually start with the normal test cases anyway. Another explanation to this is that the first task for most of the

participating testers is to do requirement testing – showing that the system works – which also shows that these test cases are ok.

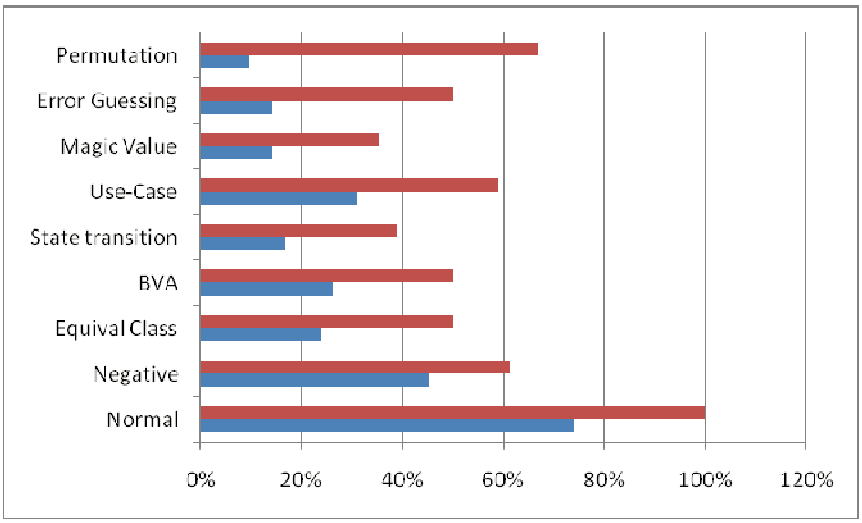


Figure 8.6 The percent of successful test cases for (blue n=42) and for all attempted (red).

Though, after our insight about systematic mistakes we went back and recoded these experiments and we could conclude that of the 31 attempts of normal TDT, and equally many successes, that almost all (but very few) normal test cases – are all the first simple obvious test case there is (confirming our result in study 12).

The techniques that was successful based on the total population (n=42) that could have attempted the technique in the given time, the % is distributed according to *Figure 8.6* (blue) is then compared with the success of % of the people attempted (red), the techniques gives a completely different picture. It is interesting to see that scrambling the order of the TDTs did not have the impact that we thought. We recorded the order in which the test cases were applied, and noted that in almost all cases, the order in which the technique was described in the "teaching" was followed. We could also see that the people that fulfilled the permutation technique, where only the persons attempting most techniques. Further analysis can be deployed on this data, but we can see how important complete data and large populations are.

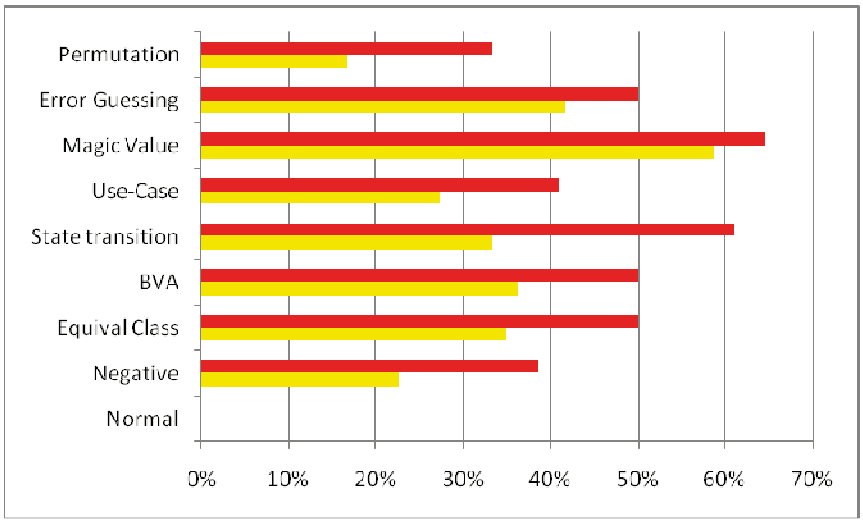


Figure 8.7 The percent of failed test cases (yellow) compared to the total percent of all failed including hesitant fail (red). This is based on number of attempted techniques.

Equally interesting is to look at the data of what we can call "over confidence" in *Figure 8.7*. The people obviously thought that they could perform the technique, and have failed in their attempt (yellow) to create a test case following it. We then apply a much stricter view on the failed data, and let all with incomplete or hesitant techniques also be a part of the "failed attempt" in the same Figure 8.7 (in red). We assume that a poorly written test case would probably get the same fate as failed, if it was in industry.

It must be a bit "chocking" to see that the most misunderstood techniques seem to be Magic Values (which is a relative easy technique) as well as Error guessing. We analyzed this further and could conclude that the reason for the high number of Error guessing is that it seems to be missing any form of description of what error is assumed and why, and that most people just randomly picked a (positive) test case. This also means that the validity of this specific technique might be lower than for example Magic Value, who was just plainly not correctly performed, but is, to our mind, a relative easy technique, once known. To our surprise, also 1/3-2/3 failed with state transition and equally one third to a half failed the equivalence class techniques and with Boundary value analysis technique. This is not good, since these are important techniques, and also -

The total number of test cases that was attempted with a reasonable test case structure compared to total number of test cases. The two most "Difficult" TDT which most people seem to have misunderstood is Magic Value and State Transitions, followed by Equivalence Class, Boundary Value and Error Guessing. This is more easily viewed in *Figure 8.8* below.

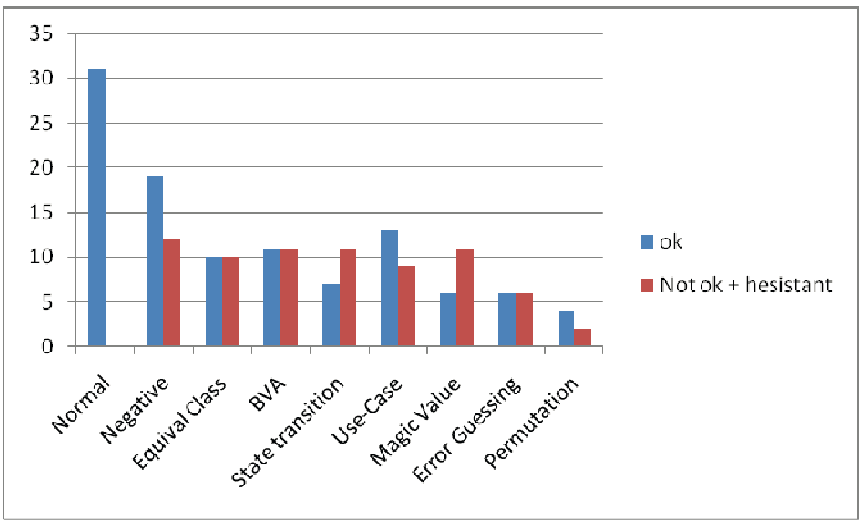


Figure 8.8 Techniques that fewer people can perform
(y-axis: Number of respondents)

In *Figure 8.8* one can really see an indication of which of the TDTs people found most difficult, but also one can see some part of "over confidence" – which makes the earlier results more interesting. It seems that we either are not being absolutely clear of the "rules" of a TDT or that the definitions are poor. In this case, we could find (at a second look) a lot of the "poor qualities" in wrong assumptions, un-detailed and ambiguous test cases make the conclusions unclear. But we do feel that there is a bit of "over confidence". People really want to look good, and it strikes us that there is a severe lack in learning the fine nuances of test cases. We often settle for the first obvious, rather than using the TDT in detail.

This brings us to the third research question: How many testers know the techniques and their application that well, that they could can understand and utilize the overlap of the most TDTs as a way to conclude the experiment? Thus produce all nine test cases within the time limit? There were 5 persons attempting to apply all TDTs. Out of these, only one person clearly managed to create valid test cases on time for all techniques, and

also – only one person realized that it was possible to "reuse" much of the same actions in the different TDTs.

The fourth research question is more difficult to answer: *Are there preferences of which techniques to attempt (and which are not commonly approached?)* We could conclude that the number one technique applied correctly and performed by most participants is the normal TDT. Equally permutation and error guessing was not approached, but we cannot draw any conclusions other than guess, so part of this question remains unanswered. The reason might have been that from the first surveys, we did preserve the order of the techniques, but not until the last few studies, we scrambled the order. We marked what order the test case were answered, and people still followed the same order (techniques they know first!)

Our last research question was: *Are there patterns on what is selected to test in the system?* We can clearly see some indications of similar patterns. This corroborates with the data for the students in Chapter 7, using the same system, Buddy, to perform the examples. Only 6% of the test cases applied had the wrong assumption about the system (based on number of TC's created), which is the number that describes the misconception about the system. This is much lower on average than the student groups had, which also shows the experience of making contextual understanding of the goal is better among more experienced testers and developers, even if the system is alien to them. This is our interpretation of the low count of misconceptions, but the evidence could have other plausible explanations. On the same note of misunderstandings, it seems like people are very hesitant to claim if their test cases are effective, which most people responded "sometimes" too. Some people stated that they found faults, so the techniques were effective. Only very few did the correct assumptions about how many test cases there was left to do with this technique, and very many were seriously understating the size of the input domain, and the type of test that was needed. Another insight is that when scrutinizing what type of faults people made, it was interesting to conclude that many confused boundary value analysis with "outside the boundary", or could not make a clear definition of what boundary really meant. It was equally interesting that many attempted and misunderstood negative techniques. We do not have a clear picture of why this is the case, but mostly, people seem to have the wrong assumption about what is a negative TDT and clearly misjudges the analysis that needs to go into this TDT. One could

discuss if it is a bit of “over confidence” in the industrial testers and developers when it comes to the know-how of TDTs, and it seems like a good investment to teach much more than the one and a half hour you need to really make the techniques a part of each individual’s personal testing skills.

8.4 Lessons Learned

This was a very valuable study. It shows that our initial assumption is wrong: It is not the case that people do know the techniques, but not the names of test techniques. It is vice versa: People are more familiar with names of techniques, but it looks like there is a vague understanding of details in using these techniques, and also in applying them. Also, the study helped us gaining a better understanding of how to perform these quasi-experiments, and the importance of thorough design of the test case.

It is interesting that during this experiment, there was one entire group of developers that did not contribute with any measurable results, apart from one person. We could only get their initial understanding recorded, thus, these people were omitted from the study. Since the application was done on their own system, it was much harder to judge if the TDTs were actually used and applied. This also shows a difficulty in the definition and the judging of the result. We must have more examples from different types of systems, and more systematic test with individual techniques must be used. It is safe to conclude that we learned that if this is to be repeated for code-level, the timeframe should be 4 maybe 5 hours minimum! It would also be good to repeat the study with 2 hours of practice (one more hour) for the nine used techniques, making sure that ample time is given for people to write correct test cases. Also, a template might aid in the judgments of the test case, since a lot of the sloppy written test cases might have been made correct if the writing was guided by a template. Now many of these were left “undecided” or hesitant.

Chapter 9. Negative Testing

9.1 Summary

Negative testing is a test technique approach that aims to target execution paths and input, outside what is clearly defined in the specification of the system. It aims at targeting the implicitly stated, the ambiguous or different fault combinations that often are left unspecified – since most requirements describe what should work in a system. The more complex the system is, the larger the combinations of possible undefined areas are. This chapter can be viewed as an initial attempt to explore the area of negative test. Only one book has been published on this topic, by Whittaker [211] that unfortunately targets specific types of systems. Our study is attempting to replicate the negative TDTs on a Telecom Middleware Operation and Maintenance (O&M) interface. Our conclusion is that many of the negative usage techniques were not well-suited for our type of system, but in general, many of our research questions are still unanswered. The negative testing area needs much more research to create more generally useful negative test techniques.

9.1.1 Context of this Case Study

We were interested to evaluate the efficiency, effectiveness and applicability of different TDTs, and were intrigued to see if this specific class (negative testing) could contribute to the industrial testing, assuming that most test cases are positive and requirement based. Our system under study was the operation and maintenance user interface of a Telecom middleware platform. We attempted to test the system through its common user interface, which is a realistic interface to work with for experienced operators maintaining the product. In particular, we wanted to check that the system could handle negative test cases and investigate if using TDTs more consciously would find new faults. Our intention was not to fictively access parts of the system, as a user with malicious intention would

attempt. This targets another area, security, which is out of scope of this study.

This study was conducted as a part of a thesis, performed by two master students [85]. This chapter will describe their work and also add on the know-how of some related work in the field, in particular reviewing negative test TDTs techniques. Furthermore this chapter particularly discusses desirable future work in the area. The intention was to work through the software attack techniques, as described by Whittaker [211] [211] which provide one of few clearly defined negative testing approaches.

Our research questions in this study were the following:

1. Is the Test Framework (from Study 2) useful in practice, even if there are known faults of the system exist?
2. Are the negative TDTs (Attacks) possible to translate from the specific environment and operating system into (industrial) middleware systems in general?
3. Would these negative attack techniques translate into any interface?
4. Would the technique be helpful in finding real important faults?
5. Could the testers (students) understand and apply theoretical know-how into valid test cases and successfully apply them?

These questions were the main reasons for this study, and sheds light on why it is important to discuss negative test. In particular, this chapter brings important information of applicability of the technique. The main finding from this study is that all the negative (attack) techniques are not straight forward to apply to our system under test, for a series of reasons. It provided us with valuable information on the difficulty to apply the TDTs on a real system, even if the TDTs are theoretically understood.

9.1.2 Design - Research Method

This study is based on the second study in this thesis – with the attempt to deploy our framework [65] (modified as in Appendix 4) for how to apply and measure the efficiency, effectiveness and applicability of TDTs. In addition to some common techniques, we focused on re-using published negative TDTs, that was originally described for shrink-wrapped, but yet middleware software, e.g. Microsoft Windows product and collected data.

These main negative techniques used were described by J. Whittaker in his book, “How to break software”[211], as examples of usage test cases that can be created to deliberately try and approach the software outside its allowed parameters and settings, thus outside its “normal” behavior with the attempt to invoke abnormal, or unwanted behavior with the software.

Our target system was the same middleware system as used in the third study of faults (Chapter 5). We selected the operation and maintenance GUI interface (since it suited the above usage techniques the best).

The TDTs were studied carefully and the system like-wise. The main work for the students was attempting to get perform the test cases of the system, but it had to be available, since there were restrictions in the access to the system and there were some initial installation problems of the sub-system, resulting in about half of the master thesis focusing on installation testing which are not included here. Once the test cases were created the students had the opportunity to test the actual target system in not only the limited test lab, but also in a proper test lab (with less restrictions imposed). The results were collected and described.

This study can be claimed to be performed at two levels. The actual test cases were conducted by two students performing the study as a part of their Masters thesis [85]. Secondly, the students were themselves a target in this study. Did the students really understand the TDTs and could they apply them correctly? The students spent an entire year on investigating and working with the software system at hand. Still these students can be considered novices to testing.

9.1.3 Validity Threats

This study is on how to examine and apply TDTs – and construct valid negative test cases. The students were supervised during the study, so there is a strong researcher bias in the set up of the study. In particular, a lot of discussions were on how the TDTs were to be comprehended. Despite this, the results do not have researcher bias since the test cases were created and results were collected independently by the students.

It became particularly interesting that the students had limited direct access to the software, the source code and the internals of the system, and could only apply the techniques through the system GUI or interface, which probably impacted the applicability of the different attacks. Limited access

in this study has a negative impact on the applicability of the attacks, but also shows that having full access and authority to manipulate the system under test is essential, but not always possible for students working on real industrial systems. More about these types of problems in applying research on industrial systems is discussed in [173].

The test cases are preserved for further study and possible replication of the study. In particular, one can discuss if they are true representatives of the different techniques and probably the number of test cases could be expanded for the non-applicable techniques. It would be particularly interesting to scrutinize the test cases with an experienced tester of the particular system, to evaluate them in the context of how they are challenging the system.

There are many threats of validity, where examples are that we used few testers, a limited target system, and that there were no control over faults and how they propagate into to failures. Another series of validity threats are due to the relatively few attempts and test cases for each technique, limitations in the system know-how and time and limitations in accessibility to the system. In its design, this is a very specific study that only is intended to provide an initial understanding of the difficulties to investigate these techniques.

9.1.4 Contribution

Negative TDTs can be defined in different ways. This chapter and study shows more the inadequacy of how these techniques are explored in industrial systems. Consequently, this could be a starting point, given the fact that negative test are instrumental for robustness. To return to our research questions 1-5, we have shown the following contributions:

1. *Is the Test Framework (from Study 2) useful in practice, even if known faults of the system (other than history) exist?*
   In particular, the first phase was omitted from the test framework and subsequently all the aspects involving relating test techniques to failures (faults). Aspects of automation were also omitted. Therefore it can only be concluded that the general approach has some merit in supporting evaluation of TDTs.

2. *Are the negative TDTs (Attacks) possible to translate from the specific environment and operating system into generic (industrial) middleware systems?*
   It is not necessarily so. From an initial standpoint, it seems that the negative techniques are difficult to apply, and only 7 of the 23 attacks could "easily" be translated into test cases for this application according to the students. This figure is probably higher in reality, since there were many restrictions in this particular study that impacts the result, but the study gives an initial understanding of how difficult these attacks are to translate to industrial systems.
3. *Would these negative attack techniques translate into any interface?* We do conclude that the attacks as written are rather specific for the applications using the particular system as described. There is a need of more general negative test approaches.
4. *Would the technique be helpful in finding real important faults?* There was a few discussable problems found in the application, but no major problems were encountered. In addition to the obvious – the high quality of the system under test – another explanation is that it could be due to the quality and type of the test cases applied. One would assume that negative TDTs do target more fault-prone area, but this particular system seems to have restricted the input variety at least in the dimension that was investigated by the students. Thus, there is no conclusive information, since the systems quality was not investigated.
5. *Could the testers (students) understand and apply theoretical know-how into valid test cases and successfully apply them?* It seems like the students had a thorough understanding of the TDTs, but the limitations in the system, accessibility, time and the test environment situation imposed on the creativity to challenge the system to its fundamentals. At least initial test cases in the area were created. Thus, the result also shows that it is easy to limit the actual application of the TDT into test cases for negative tests (fault test) and equivalence partitioning. Thus, it is not straightforward to create examples for these techniques. We could also conclude a general misunderstanding that the attacks were not viewed by the students as a sub-group of negative test, thus negative test had its own "results", separated from the different attacks. That makes the view of what is a negative test rather obfuscated.

9.2 Introduction to Negative TDTs

Negative testing is a group of TDTs. This area of testing focus on getting robust, stable and high quality software, and the level of robustness depends on the business or system domain. If we aim at completely eradicating the possibility for any faults propagating into a system failure, e.g. in safety critical systems, this requires a series of methods and techniques and it is a very costly endeavor to create a so-called foolproof systems. There exists a wide range of how much effort you are willing to spend to mitigate the business risk and minimize the consequences. E.g. internet software that involves money or sensitive information should be very well tested, whereas some systems can tolerate small problems, if there are ways to work around them, or if they get fixed within a reasonable time.

9.2.1 Negative Test

Well-defined and well-specified systems will have less use of the negative test approach, and on the other hand negative techniques are more successful in systems with more undefined specifications or if the complexity of the system is high. A series of software, e.g. complex systems, the majority of open source systems, and systems which are not directly having a "open to customer or public" GUI/interface, are for different reasons often left with areas that to a high degree assume that the user knows what they are doing.

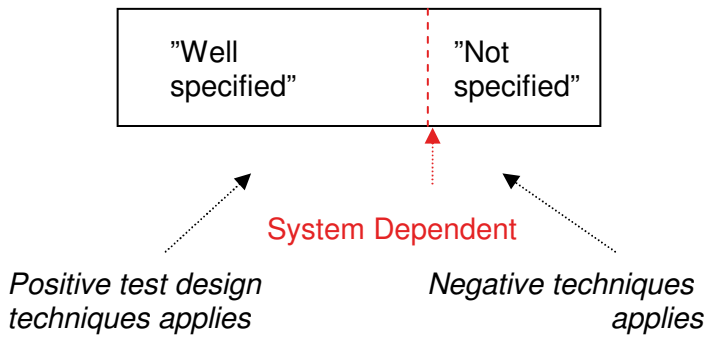


Figure 9.1 Specification level of the system behavior implies TDT type

We have described some of the negative TDTs in Chapter 2.7 (additional to the use in this study). In this chapter we first describe the negative TDTs, and then present the specific study. We will, during our description of each of the techniques used also relate it to our techniques taxonomy, and thus hint at overlaps of the techniques. Examples of such techniques that contain negative (or outside the specified allowed input) are equivalence partitioning and boundary value analysis. Another negative techniques that is often useful is permutation, since it targets software that will allow unspecified or implicit order of execution, which can cause problems.

9.2.2 Related work on Negative Test

The negative TDTs are not yet fully explored. One can probably define specific variations and combinations, and name them as contributions of future work. In fact there has not been enough focus on negative TDTs, so when Whittaker [211] came with his publication based on usage testing and analysis of shrink-wrapped software (Microsoft) usage systems, it was a strong contribution. Since by sheer number of users, these systems must be tested to almost "fool-proof" level, they also provide a good study object with a large collection of available faults.

9.2.3 Negative TDTs

In our study the following negative TDTs were used, and this list below is an expansion of the Master Thesis [85] and thus the original book [211]. Thus, having the deepest respect for the author, Dr J. Whittaker and the copyright of his book, we are not explaining the full picture of each of these so called "ATTACKS" that in most cases can be translated into useful TDTs. We are describing some of these attacks with a different depth to be able to contrast them in our test design Taxonomy and for scientific purpose aim to identify overlaps, to further enhance this important set of techniques. Another contribution in this thesis is the attempt to expand the "Attack" into a more general approach working on all types of software. As we can conclude, this is probably not realistic for all of them, and especially difficult are the ones that are done in combination, and that approaches test from different aspects (including malicious behavior).

Attack 1: Apply input which forces error messages to be displayed

This "Attack 1" according to [211], is an assembly of techniques aiming to address error-handlers that are difficult to get right. This is done by using input that is outside the valid or normal range, i.e. too big, too small, too long, or too short, or outside acceptable ranges or values of the wrong data type. If all error messages are defined, it should be possible to define a specified positive test to reach it, thus at all execution threads that possess an access to a fault handling (generic) or error message, the particular test case to invoke that should be accessed. In particular, Whittaker [211] specifies the following variations:

1. Input type (give "wrong" format, e.g. for expected integer input, give real number or character) (see also generic Attack 3 below)
2. Input length: Too many characters, or null (see also generic Attack 4 below)
3. Input Size: Too small or too large (for text, implicit of 2).
4. Boundary Values (e.g. based on glyph, the representation of the ASCII system)

Our conclusion after describing our structure of test cases is that it is possible to conclude that 1, 2 and 3 in the above list are both variants of negative input variation, and that we will see an overlap (within the attacks) of the actual TDT, but the target and circumstances of applying them are different. The Boundary Value check (4) is a variation for e.g. different representation, but is also a part of a specific technique. The concept used, does not take execution paths into account, and focuses on "user" usage input. There is no restriction that this concept could be used at any domain and/or level of software, but is (see p.21 [211]) assuming development layer access to the software. The limitation is when design by contract is enforced, and it is clear that the checks for sending correct input are taken care on another abstraction-level in the design paradigm. Yet, any type of fault-, error- or exception-handling code are assumed in general to be more error-prone, a claim we have not been able to show, or find supporting evidence for – but the reasoning seems viable, since it is often difficult code to test, and is seldom a focus or priority for a test project.

Attack 2: Apply inputs that force the software to establish default values

Whittaker [211] claims that main idea behind this attack is to force software to establish or use data which have not been initialized with default values, following a correct flow of variable declaration, initialization and usage. This attack aims to test all such test cases and is applicable to any program that declares and uses initialized data. The issue being that this definition is again incomplete as is, because the test is not only checking the established default values per se, but furthermore to challenge the system, by using these values in combination with other aspects of the system that could cause problems, especially if the system re-initialize values, and have specific checks. One example could be letting the system set the default values, and then using that value in a calculation, and then change, re-use or manipulate with the default value by giving another value etc. Thus a series of variants of this technique could probably be contextually described depending on the software application under test. The main problem of this attack-group is that it proclaims an aim, but cannot define a specific systematic repeatable approach, thus is more of a "search quest" than a technique. Thus, usage of established default values could be more important in some systems than others. Static analysis tools can easily identify unused variables, redundant declarations, and some (but not all) aspects of lack of initializations. Thus finding ways to bypass through execution the intitialization step before usage can be an intricate problem in some systems. From a schematic point of view we would like to define this either in the section "positive – valid" category, with the variant of enforcing default values. The other aspect of this technique is to use this in combination with aiming to address and bypass checks. Thus, we deem this Attack really into two different techniques, one to establish default, and then as a "combination" TDT, "default in usage".

Attack 3: Explore allowable character sets and data types

This Attack as per Whittaker [211] can be viewed as an expansion and completely overlaps the first item in the list in Attack 1, that was limited its cause to the aspect that it should force an error message. He defines three aspects where this is interesting, where all of the character sets and data types are to be considered whilst doing input analysis. Thus it should be well defined what is allowed and what is not allowed for a specific system, thus the Character sets (ASCII or UNICODE) specially handled by

different operating systems or by different usage in Programming languages.

The character sets have ordinary members and special members, and so Whittaker advises testers to follow a table (Table 2.2 on page 29-33 in [211]) developed by one of his students to apply this attack. These categories are not fully explored and give an indication of the complex problem at hand. In particular, he clearly defines the applicability of the Attack and suggests what sort of expected behavior that can be defined for each of these values.

Attack 4: Overflow input buffers

This attack exploits the fields where there are no constraints on the length of the inputs, which the software can accept, according to Whittaker [211]. Long strings and too much data can crash the software if not handled properly. This attack basically occurs when developer fails to consider the string size, which is supposed to be passed into the memory buffer. This technique is again an expansion (or overlap) with the second and third item in the list in Attack 1 that limits itself to values that has a purpose to "force an error message".

Attack 5: Find inputs that may interact and test combinations of their values

This test approach aims to locate faults that come into the picture when different combinations of inputs and the relations between them are tested. This attack aims to create scenarios such as when one input affects the value of other inputs, thus attempting to find related inputs that describe aspects of a common internal data structures, and particularly using them in computations. According to Whittaker [211] this is a very valuable attack for testers where multiple developers are working on a same code base. In particular, this should be a part of a normal input analysis that is not only concluded in the one dimension, but also is defined for multiple dimensions. Here often combinatorial explosion happens in theory, but in practice, there might be combinations that can never occur, posing difficulties when analyzing reachability to all parts of code in the software.

Attack 6: Repeat the same input or series of inputs numerous times

Whittaker [211] suggests in this attack to enter the same input to the system repeatedly, as this behavior generates the chances of exploring faults which

generally pop up due to memory leaks and poor exception handling. This is a technique that benefits from automation. Repeating test cases over and over is standard technique – that particularly involves filling buffers, exhausting the system in different ways, e.g. different forms of saving, or exhausting local memory or some limited hardware, and challenging other resources of the system, such as timing, memory and race-conditions. This becomes a specific technique in itself. This can also be expanded on a much grander level, where the system repeatedly can be filled with information and used in regression test suites.

Attack 7: Force different outputs to be generated for each input

This attack is performed to show all possible outputs related to each input in the same circumstances – or to apply a test with as many different contexts as possible. The idea behind this attack is to understand the behavior of software when a user applies a particular input and system generates different output based on prior inputs. It is important to clearly define (enumerate) all "variants" of output, one (or a combination of one set) of inputs can have. One could say this technique explores hidden dependencies and earlier actions that are not taken into account when performing the new "test case" or action/execution. It is possible to claim this is a lack of having proper assumption of previous possibilities when commencing the start of the test case. Defining a set of contexts, configurations, input combinations possible and listing its associated output can be costly, and is often difficult from a user point of view (at a late stage), and thus is a great source of faults to rectify when specifications are inadequate.

Attack 8: Force invalid outputs to be generated

This attack makes a list of all possible wrong answers, and check whether by varying input parameters; a tester can get those results. This test technique assumes prior knowledge of known problems with software, and could be assumed to be a variant of "error-guessing". Since there is no particular guidelines for generically "known" mistakes and wrong answers, one can only speculate for any software what "wrong answers" might contain. Understanding what a correct output (value) is from the intention of what the software should produce versus what is stated (in specifications and documentations, and – in the actual system), cannot be defined into a specific TDT, but more be seen as a general know-how of the system.

Attack 9: Force properties of an output to change

Most outputs from a computer program are non-editable, but for all output that can be changed in any way, e.g. color, shape, dimension, size or any other attribute, it is possible to end up with an internal inconsistency that can cause malfunctions. This attack seems to be more adequate for graphical user interfaces, where the user is allowed to change the properties. From an abstract point of view, one could claim this attack is a variant of "too small, too large" (third item in list in Attack 1) or Attack 4 for the graphical properties (and other negative attacks for other attributes), using specifically output, and the graphical handling combined. Furthermore, in Whittaker's proposed example, he includes repeating the approach several times, varying the attributes back and forth, which is Attack 6. Generalizing the attack to any type of editable output makes the attack more interesting than only changing its attributes. Is it possible to change outputs in many systems, and thus, in what way? The question we pose is: Can this be turned into a defined TDT? Is the technique to create test cases for each editable output, and approach it with both an analysis of attributes possible to change, and then making a repeated variation of these attributes using appropriate aspect of change (e.g. too large, too small). This seems to be a combination technique specifically targeted for editable output.

Attack 10: Force the screen to refresh

The attack is aimed to create and modify objects on screen, which eventually makes the application or operating system refresh the output screen. As a result of which the output screen fails to refresh it and objects on the display might appear garbled. The very frequent refreshes in the window may cause minor delays or major problems.

Attack 11: Apply inputs using a variety of initial conditions

Whittaker [211] explains this methodology by suggesting that a feature of software or function should be isolated, and all data objects that are supposed to be used by the feature or function are to be considered. Data objects should be partitioned such that all data objects in a partition show almost the same behavior and tester should execute at least one combination from each partition.

Attack 12: Force data structure to store too many or too few values

This attack is designed to explore whether checks on array boundaries, lists, and other possible data structures are in place or not. Lack of proper coding can produce underflow or overflow situations, or can cause corruption of data. This is in principle the same test technique as third list item in Attack 1, but with a different target, thus is deemed as a variant.

Attack 13: Investigate other ways to modify internal data constraints

This type of attack is used to check whether the programmer has coded the error handling mechanism for not only at the time of creation of data, but also when the data is modified or accessed. This cannot be translated to a specific TDT and can be claimed redundant to Attack 1, (2), 3, 4, and 12 but specified or made for any data structures. Since "investigate" is a bit too generic, we are not claiming this to be a unique TDT, but a variant at a different level. Again, we conclude this as a variant, and do not become a specific technique in itself.

Attack 14: Experiment with invalid operand and operator combination

This attack explores the computations where operators and operands are involved. Thus, specific values used in computation have meaning. In principle, the attack includes repetition of computations several times, and attempts of reusing too big, too large in the context of values (numbers) in the examples. The author also claims that it's an effective technique for those applications where graphical rendering happens. This would only restrict the usage to graphical handling. The more generic approach is exploiting values using computations or in a series and combinations of computations.

Attack 15: Force a function to call it recursively.

The goal is to be able to force a function to call itself recursively. The idea behind this attack is that an object may communicate with other objects easily; however when an object is supposed to interact with itself then an error may pop up. This attack exploits the recursion. A very common example is a web page which has a link to itself.

Attack 16: Force computation results to be too large or small

This attack aims to reveal problems by creating the scenarios of underflow and overflow of data objects which store computation results. The most apt applications for this type of attack are those in which computations are performed very frequently and results are stored internally. This is clearly a variant of Attack 1 and Attack 3, and also using repetition (as of Attack 6) and using specifically exhaustions of storing types and particularly addressing computation results (or intermediate results). Thus this attack is deemed a variant.

Attack 17: Find features that share data or interact poorly

The idea behind this attack is to look for features which share data or resources with other features, and so might cause another feature to break. This can be achieved by allowing the common characteristics or resources to be identified, and then a dead lock scenario is created which might lead to anomalous conditions, furthermore to "get in the way" graphically to each other. In one aspect this might be considered unique if only viewed as a user or GUI system attack. If generalizing this attack, two features interacting (in any way) is in principle a target. Thus, defining what is a "poor" interaction could be defined by scoping the parameter communication, by defining dependencies between these features etc. This will soon become a very complex analysis, since dependencies goes both directly and indirectly. For most software, static analysis tools are an aid in understanding some of the relations. Translating this to a ubiquitous TDT seems fruitless. It seems like the variation of shared resources is not fully explored in testing, and purely looking at it from how it is displayed in different user interfaces is not sufficient.

Attack 18: Fill the file system to its capacity

This attack is based on the belief that if the hard disk of the local system, the database, the stack, heap, local memory etc. is full, then the application may behave in strange ways, e.g. operations like open, close, read, write and modify etc.

Attack 19: Force the media to be busy or unavailable

In this attack Whittaker [211] advices testers to check for error conditions related to the storage device. The idea behind this attack is to force the error return codes which indicate the problems when applications access the media devices such as hard drive, floppy drive, compact disks, or other

external storage devices. The author suggests creating a scenario where the application tries to access these devices, and then making them busy by creating the condition of contention between different applications for the same device. This could be viewed as testing or checking of the software in the context of environment. It particularly would handle the interaction with faulty or malfunctioning hardware or other parts of the software (in a complex system).

Attack 20: Damage the media

When testing of fault tolerant, fail safe or mission critical software the tester should try this attack by "damaging" the media, as e.g. the software are often designed to work even if the media is damaged. This attack aims to ensure such an error handling mechanism exists and performs necessary actions. In general, this could be abstracted in many different ways, such as damaging different parts of hardware, loosening cables, and similar actions. This attack is just a stronger (or more definite) "grade" of unavailability than the previous Attack 19.

Attack 21: Assign an invalid file names

This attack can be performed specifically on the combination of older operating systems file-handling, which often have specific restrictions and on applications where no constraints exist when trying to read from file identifiers and write to file identifiers. Failures can be caused if the restrictions are not placed on the application level. The attack is simply attempting to open, read, write and save a file with a name that breaks the rules for the operating systems restrictions. This can be viewed as a dependent, combinatory problem (or integration issue), whereas the two different boundaries should match. The attack is really checking that the interface has safe use of the parameters – in this case – the right "restrictions" of the files, thus attempting to test outside the boundary of the operating system.

Attack 22: Vary file access permissions

This attack lies in multiple access or change points for file permissions. This attack focuses on altering the file access credentials and forcing the application to access or modify the file contents. One example that can cause failures is if one application opens a file (application has read only permissions) and another application is trying to write on it (which has write access permission). The test technique is particularly finding

mismatches on a level where a file can be opened by different applications or different versions – that has mismatching access. From a test technique point of view this is really fulfilling the functional requirements, by systematic confirmation of the file-handling permission properties.

Attack 23: Vary or corrupt file contents

The idea behind this attack is to modify the contents of input files intentionally. If the error code has not been in place to check the file contents before they are read, the software might crash. This is really a fault-injection technique, to double check that files are properly checked before coming into use, since it is a relative common problem that files get corrupt for different reasons, and this should be handled well by the system.

9.3 Structuring Attacks

To better understand the attacks that are specific in many aspects the techniques we have structured them in two ways. In Table 9.1 we summarized the different attacks and distributing them over a series of conceptual TDTs, in fact, the techniques could be collapsed further.

Variations of the following manners could be grouped into defining what is within (allowed) the correct attributes, and what is outside the defined area, and consequently should not be allowed. How the “not allowed” could be outside the boundary, using malicious input or output, or a series of “malicious” attributes, should result in different behavior depending on context, e.g. invoke either a fault handling or other fault tolerant behavior.

Examples of such variations could be structured like this:

- Input variation
    - size (length, bytes)
    - character sets, types
    - based on output
    - based on combinatory (input, output)
- Permutation
    - input
    - order
- Fault injection

- Repetitions
- Dependencies where two or more aspects are involved:
    - hardware, environment, context with the software of different attributes (e.g. permission…)
- Combinations of (any) of the above

Table 9.1 Distribution of Attacks over TDTs

TDT	Size		Type	Character sets	Outside BV	Order Permu-tation	Repetition	De-fault	Dependencies combinations	Refresh GUI	Context Env
Attack	1a	1b	2	3	4	5	6	7	8	9	10
1.	x	x	x	x	x						
2.						x		x			
3.			x	x	x						
4.	x	x									
5.									x		
6.							x		x		
7.									x		
8.				x	x	x					
9.	x						x		x		
10.										x	
11.									x		
12.	x	x	x		x						
13.	x	x	x	x	x			x	x		
14.	x	x			x		x		x	x	
15.	x	x			x		x		x		
16.	x	x	x		x		x		x		
17.	x	x				x	x		x		x
18.									x		x

TDT	Size		Type	Character sets	Outside BV	Order Permu-tation	Repetition	De-fault	Dependencies combinations	Refresh GUI	Context Env
19.											x
20.											x
21.	x	x			x				x		x
22.									x		x
23.									x		x

It turns out that many of the negative attacks are specifically focused on know failures suited for the system under test. To avoid losing the complex detail of these attacks, we are also showing in Table 9.2 the different distribution of Attacks (based on the Attack number) over specific targets. As seen both the Table (9.1 and 9.2) are related.

Table 9.2 Distribution of Attacks over targets & type of technique

	General properties	Error messages Fault handlers	Input	Output	OS Files	Prog Lang	Data struc-tures	GUI	Inter-opera-bility context
1a	Size (too big or too small)	1.3	3, 12, 14	9	21		12, 13, 16	9, 14	17, 18,19
1b	Size (too long, too many chars, too short	1.2	4, 12, 14		21		12, 13	14	
2	Type	1.1	2, 3		3	3	16		
3	Character Set	1.4	3		3	3			
4	(Outside) Boundary Values		12, 21	8	12, 18, 21, 22		12, 13, 15, 16	14	
5	Order/ Permutation		2						

6	Repetition		6, 14	14	18		14, 15, 16	9, 14	
7	Default		2, 11	9			13		
8	Dependencies combinations		5, 7, 11,14, 16, 17	7, 14	11, 18, 22		13, 14, 16, 17	17	17, 18,19, 21, 22
9	Force to Refresh							10	
10	Fault injection (hw, env.)				23		23		19, 20, 23

9.3.1 Negative Test for Industrial Systems

Having reviewed the different attacks and concluded that there is a great need to better structure negative TDT that could be generalized for all types of software, we here propose a new structure. This structure is a derivative from our review of existing techniques and is neither verified nor investigated in depth. The goal is to ease the view and understanding of the field of applying TDTs. The principles could be described in the following manner:

1. Negating "specification" (e.g. requirement)

- For every specification statement, it should be possible to define one, or a series of aspects in this statement that can possess the "opposite" state.
- Make a matrix or list for each of these items (allowed, not allowed)

2. Failure scenarios "What- if"….

- For every "use case" description, define one – or a series of "what-if" descriptions, and the consequential behavior of the software for those situations
- Explicitly state assumptions about the software (post-condition) and define solutions for "what-if" these assumptions did not hold

3. Make a thorough Input analysis

- All input states, make specific groups that bears meaning:
    - Default values
    - Character sets
    - Types
    - Size (too big, too small, too long, too short)
    - Values of input that have special meanings
        - Integer, float, exponent
        - 0, null, negative numbers
        - Operating systems characters
        - Boundary values taken all above different aspects into account
- All input states with dependencies (to output, other input, computation or any other attribute)
    - To time (first time, second,… 1000nd time)
    - Time dependencies (order, permutation)
    - Memory usage, stack, heap, buffer sizes, file sizes,
    - Any other hardware, third party software
    - Operating system dependencies (e.g. unique file sizes etc).
    - Attributes (e.g. permission)
- All different actions for "disallowed" input
    - Make sure to define fault-handling, restart, or recovery
    - Corrupt environment
    - Poor parameter range, size (in interactions)
    - Specific software faults (injected) or imposing hardware malfunctioning (e.g. pulling cables, cards)

This structure can aid in creating test cases that supports a robust system, and verifies different aspects of the system at different levels. With the aid of these TDTs, one can define different robustness levels and principles for the software under test.

9.3.2 Introduction to the Industrial Case Study

This Industrial Study [85] contains several elements. Not only did two students deploy existing TDTs on telecom middleware software, with the intention to evaluate the applicability of these techniques, but also the

students themselves were the target of research. We were curious to measure if the students had fully comprehended and understood the specific nuances of the different TDTs, and could they transform this knowledge into a set of test cases, that did challenge the software. They were responsible for learning and studying the techniques, and used a rather limited set of system access privileges to conduct the test cases.

The result is intriguing in many aspects. Not only were a series of these techniques not possible to perform on this operation and maintenance interface, but they did additionally only contribute marginally to the fault finding. This indicates that the approach taken in Whittaker's book [211] has serious limitations in applicability, when in use for these types of systems and that more studies must be performed to improve the area of negative TDTs to better target the software under test. Secondly, the unexpected low fault finding ability could indicate that the software under test is of good quality, thus one must be careful to draw any major conclusions on the effectiveness of the techniques used. Thirdly, we could conclude that the students had seriously misunderstood and confused some of these techniques (and their possible applicability) in this context – which affected how they could deploy them. This could be visible when scrutinizing the test cases and gives an indication of the difficulty. Applying theoretical testing know-how into factual useful test cases that has meaning for the system is not an obvious and simple task, especially when you are unfamiliar with the system.

9.3.3 Techniques used in the Study

The 23 attacks as described by Whittaker [211] and the following TDTs were attempted in the study by the Master Thesis students:

- Positive tests (in thesis called "normal tests")
- Equivalence partitioning
- Boundary Value Analysis
- Negative test (in thesis called "faulty test")
- Random Input variation, where a tool was created to randomly pick from entire input domain (ASCII/glyp)
- Fault Injection

In the result section we will expand on the students' view of these techniques and their usage, and then comment on their comprehension. We will in particular discuss the application of the TDTs, since the test cases were available for scrutiny.

9.3.4 Target System and Approach

The system under test was the same telecom middleware system used in Study 3 (in Chapter 5) that is a large and complex system with many components. As the target software for this study, we decided to take the operation and maintenance sub-system that provides the following services:

- Management services support
- Different management interfaces on IP based Protocols
- Management application support

The GUI is developed in Java. The operator uses a remote machine (with a Windows or UNIX based operating system) with a web browser in which a Java virtual machine is also installed. The browser connects to the node and downloads and executes the developed management system, through which the different functions can be accessed. A lot of the inputs are expected strings or integers, in addition to a selection of allowed navigation characters like TAB etc. When looking into the systems features, it becomes evident that there are many limitations of the GUI imposed, to minimize the input but still provide all essential management aspects of the system, including fault handling. What is accessible and can be handled give but a minimal insight into the complex functionality underneath.

During the set up of the study, the students had a series of problems with the installation, and the installation procedures, and found lack of adequate documentation, and combinations of hardware and software that were not tested. In addition to this study, a series of problems around how to perform installation resulted in an additional twenty-two installation test cases, which are discussed in the Master thesis [85], where also the system under test is described in more detail. The system had a user interface that in the majority of its inputs from the user was handling input as strings. The motivation for this is that the user for this system is trained in the usage, and assumed knowledgeable. Only for a few places, specific checks were enforced, and thus the students had to search very hard to create test cases

that would find these specific checks, since the documentation in this aspect was missing.

9.3.5 Results Measured by the Students

The students' defined applicability as their ability to create a test case for the particular TDT as described, for the system under test. When they failed to create a test case or it made no sense to do so, they ruled the technique as "non applicable". This poses in the context of research a difficult question. Should we trust that this is valid, or was this just a limitation with these students? We had some indication that some of these techniques were imposed with limits, and might have been applied wrong.

Another explanation is that the student had a special interpretation that might not be the main intention, but can also serve as a feedback on the descriptions and difficulty of the individual techniques. We will discuss this further in the lessons learned study. Thus, the main flaw of validity is that there were only two students performing this particular task. The experiment should be repeated with a different set of testers.

Table 9.3 The students' perception regarding applicability of TDTs

	Name of technique	**No of Test cases**	**Applicability**
1	Normal tests	27	Applicable
2	Equivalence class partitioning tests	23	Applicable
3	Boundary values analysis tests	19	Applicable
4	Negative tests	23	Applicable
5	Random input tests	20	Applicable
6	Fault injection tests	2	Applicable
7	Apply input that forces all error messages to occur	9	Applicable
8	Apply inputs that force the software to establish default values	-	Not Applicable

	Name of technique	No of Test cases	Applicability
9	Explore allowable character sets and data types	1	Applicable
10	Overflow input buffers	2	Not Applicable
11	Find inputs that may interact and test combinations of their values	2	Applicable
12	Repeat the same input or series of inputs numerous times	1	Not Applicable
13	Force different outputs to be generated for each input	3	Not Applicable
14	Force invalid outputs to be generated	2	Applicable
15	Force properties of an output to change	-	Not Applicable
16	Force the screen to refresh	-	Not Applicable
17	Apply inputs using a variety of initial conditions	-	Not Applicable
18	Force data structure to store too many or too few values	-	Not Applicable
19	Investigate other ways to modify internal data constraints	2	Applicable
20	Experiment with invalid operand and operator combination	-	Not Applicable
21	Force a function to call it recursively	-	Not Applicable
22	Force computation result to be too large or too small.	-	Not Applicable
23	Find features that share data or interact poorly	-	Not Applicable

	Name of technique	No of Test cases	Applicability
24	Fill the file system to its capacity	-	Not Applicable
25	Force the media to be busy or unavailable	-	Not Applicable
26	Damage the media	-	Not Applicable
27	Assign an invalid file names	2	Applicable
28	Vary file access permissions	-	Not Applicable
29	Vary or corrupt file contents	2	Applicable

The students comment their test techniques in the context of the application like this (some editing done to improve the presentation):

"We designed 27 normal test cases based on the design and functional documents. This system has a huge potential for designing normal tests, as the system has many fields which can take inputs. We chose to focus on the mandatory fields and some important functions in the application. We applied 23 test cases based on Equivalence Class Partitioning. Applying this technique proved to be tricky as the fields in the application was accepting a wide range of inputs, due to the data type of the fields being declared as a string. We then designed valid partitions being alphabets, numeric, alphanumeric, special characters, and key combinations of other special keys such as Ctrl+C, Ctrl+V, Alt+Tab, Esc. For Boundary Value Analysis, 19 test cases were applied. The test cases for this technique made use a lot of the partitions we created for the Equivalence class technique. We applied these test cases in fields having certain checks on inputs, and also in fields which accepted IP addresses as inputs. We applied 23 test cases designed using Faulty (Negative) test cases on the system. This technique was relatively easy to apply, and the large number of fields in the application increased the scope of applying a large domain of negative inputs. 20 test cases were based on Random Input.

These test cases used the random alphabet and random number generator to generate input values. These values were then applied in some

mandatory fields of the application. We designed 2 test cases based on Fault Injection. Identifying an area to apply this technique on the system level was an uphill task. After some investigation, we found that we could inject faulty code in a XML file, which acts as an input to the system. We then executed 2 test cases which fed faulty XML files to the application. Restrictions in access to the code of the application hindered our efforts to properly apply this technique. We designed 24 test cases based on the attacks which were feasible for this system. We found that the attacks based on the user interface are most applicable, where as attacks based on the file system interface were very less applicable on the system under test. However there were also a set of attacks which were not applicable at all, as they are suited best for applications which are more interactive, and have a superior graphical interface than the chosen system."

One can conclude that the interpretation of what consist of a negative (faulty) test case is in general correct (interpreting faulty input as non-expected input), but is limited, since negative test cases can be a series of other approaches, in particular, all the different attacks can be considered to be negative test case approaches. Similarly, they clearly understood the wide bound of what is meant by equivalence partitioning, but did only provide input as different numbers partitions.

Initially the study collected timings for the different aspect – but similar to the next study (that was conducted in parallel) little or no difference could be detected of the timings of the different test cases. The efficiency of the test design construction disappears, since the main time seems for these systems to be spent on trying to understand the system (as well as the techniques).

9.4 Discussions and Lessons Learned

It is in particular interesting to note some of the particulars of the collected data.

First, what was considered table entry 4, "negative" tests, is probably a bit misleading, since most of the attacks are also negative tests. For table entry 8, it is interesting that the system contained no known default values. This might be a lack of time or access for the testers/students to look through the source code and documentation to establish such values or it might be true that no such default values exists at all for this particular software, hence

the interface could not provide any – and input was expected to be provided for all entries. Many of the inputs were either a string or integer in a specific format for this interface, which might be a valid reason for this absence. A second attention can be put on the areas where test cases were created, but the test technique was ruled as “non-applicable” as true for the entries 10, 12 and 13. This could be due to confusion by the students of what applicability really means. It could be that the test cases were not possible to execute, once the system was available, or that attempting these test cases failed in some step or made no sense. One must then discuss the aspect of the test case. Was the test case correctly written? Was it written before the availability of the system? The latter is most probable, but then the test case should have been changed or adapted. This also sheds light on how difficult it is to target the real problems while doing detailed research of TDTs.

We can conclude that it is difficult to adapt the attacks for at least some specific types of software in the industry. Only 7 out of 23 Attacks could be applied according to the testers/students. In this particular case much of the input lacked controls since not only is it assumed that the user of these interface are familiar with the system and would name things correctly, but also that most of the inputs are handled as strings, lacking many aspects of the specific input controls that would be frequent in a shrink wrapped application. One can also discuss what the correct level of checks for different type of user interfaces is, but that discussion only influences what test approach should be used. This adds on the insight that the level of negative test should be dependent on the robustness expected of the software. It seems that the most limiting factor here is the access to the system in all aspects e.g. attacks that assumes “privileges” to the file/operating system (damage the media, and change permissions for example), but in particular the internal access (call something recursively), such attack would require a much deeper understanding of the system and relations. One could call this a limitation in time that would make it possible to create a number of more intricate test cases or it could be a case of time to gather a deep understanding which might be necessary for constructing and challenging the system. One must learn more by reading and understanding the system in its context, reading the source code, learning about fault history, and getting experience and know-how of this particular system to construct really challenging test cases beyond the ordinary. Unfortunately, testers in industry must also possess motivation and interest to pursue this learning, since there is seldom time in the

ordinary work to create test cases for some of these negative approaches. It is easy to get the notion that most of these negative attacks are not naturally simple to apply.

Chapter 10. Open Source Testing, TDT Applicability and Complementary Coverage

10.1 Summary

The tester's main challenge is to construct as effective test cases as possible, given the limited time and other resources available. A key issue in meeting this challenge is the selection of appropriate TDTs. In this study we introduce a new way to measure the efficiency of TDTs, and look further on the applicability of these techniques. To make the results more general, we used 15 small systems in our study, representing different types of applications. We wanted to explore the level of testing achievable on a system with limited test effort, as well as how easy it is to apply a TDT on a given system and further explore the use of coverage as a complementary technique.

We introduce coverage as a new way to measure the efficiency of TDTs, separated from the efficiency of the test case execution. This distinction has great impact on what to measure and how techniques can be juxtaposed. Applicability of TDTs is specific to the system.

We conclude that there is strong advantage to evaluate the coverage as a complementary method for comparing functional TDTs, since the most added value is gained by exploring areas that are not earlier tested. An interesting observation was that the different coverage test tools did report different coverage measurements for the same set of tests and system, which put important questions on how coverage is defined and measured, and how these tools are instrumented.

10.1.1 Context of this Case Study

The motivation of this study was to investigate the efficiency, effectiveness and applicability of a series of TDTs on a series of different systems. We

wanted to build on our test comparison framework in Chapter 3 [65], while excluding the fault-understanding required in that process. The process had two iterations. We first selected and tested all the open source systems based only on the functional TDTs selected, and in a second iteration over the same systems we used complementary coverage techniques. We learned a lot by this approach, the most important lesson being that the added value of similar (overlapping) techniques are related to the order in which they are applied, since it is relatively easy to find the first couple of test cases, but after a while is a small added value in varying the input, since the tests in fact exercise the same execution paths. Inventing completely new test cases in new areas of the software is always increasingly difficult. We also learned that a systematic approach has a lot of merits, and were surprised by the new view and insight the use of coverage tools brought to the testers. Completely new test cases could be created, which also allowed for more fault finding. The study was performed over a time period of a year.

10.1.2 Design - Research Method

Our approach is to sample a series of test cases that is systematically applied on the system. We assume that this is a fast and efficient way to get an indication of the quality of these systems, where finding one fault might be a lucky coincidence, but finding two or more, and doing so quickly, would definitely indicate poor quality. We have also evaluated if there is a pattern visible in the different TDTs used, if any one of them are less applicable, and if so, the reason for this. Then we continued to find a coverage tool that would work for these systems, and out of our initial set of six coverage measurement systems, we got four of them working for some of our systems. Our initial try revealed that the tools instrumented the source code differently, thus resulted in different measurements. We were intrigued by this difference, and selected four of our systems to make more in-depth study of coverage, since test execution will be substantially more time consuming for the testers – not having access to automated test execution tools.

The testers did treat each of the selected system in the same manner. First a learning period was set up, to familiarize themselves with the system, then a series of test cases was created using the different TDTs, and the time to create each test case was recorded, in addition to fault and coverage measurements. The final stage was using the coverage measurements to

improve the test cases, by studying the source code. The test cases were then executed again to obtain new coverage measurements. During this time, the testers also recorded some of their experiences. The advantage with having the same testers testing a lot of different domains is that the manner of how the test case creation is performed will remain essentially the same. This gives an important feedback on which techniques that found most faults, and also which technique that was perceived effective for the testers.

The design of this study was similar to the study in Chapter 9, i.e., as a two-tier empirical investigation. With this we mean that we did not only study the system under test, we also looked at the TDT, the process, and finally on the testers approach and comprehension of the approach. Details of part of this functional study can be found in the Master Thesis [171] by the two students who act as testers in this study. This set up has the advantage of no researcher bias in the data collection and data aggregation.

10.1.3 Validity Threats

Potential threats to the validity of any empirical studies are many, since it is quasi-experimental by nature [215], which mostly depends on the representativeness of the applications, and the people involved in the data collection. The threats of external validity are therefore many. Our study is no different than most, and since it uses a biased selection of small systems, probably not representative for the entire class of software systems. To our favor we have at least enough collected data and 15 systems, which give us some significant results. Nevertheless, the lack of randomness makes statistical methods a hesitant approach; instead we have focused our efforts on more descriptive results. Another bias is the selection of test cases, since another person (tester) would probably select different test cases. This indicates a need for replication in a larger setting. We assume that the TDTs were applied correctly. Another threat to validity is the problem that the random input is simple in its selection of data, and it can thus be questioned if it is being truly random in its execution. We assume it has been sufficiently random for this study.

The internal validity has had the testers believe that the TDTs are orthogonal or unique in some aspects, but it looks like most of the techniques are strongly overlapping. They are typical examples of how

different goals and approaches can lead to different test cases, but it would be possible that a minimal number of test cases were used to cover the different techniques. The testers made an effort to find "new" test cases (and thus improve on coverage), which seems to be a rather common approach of testers. The construct validity regarding the timings is reasonable considering the circumstances. A relatively inexperienced test team, similar to our testers, would use the same approach on the same software and would most likely end up in similar time bandwidth (between the dashed lines in Figure 10.2 and 10.3). The advantage of the overlapping test cases is that the data validity is strong. The variation around 2 minutes is probable, and the only test technique not following the pattern is fault injection (which includes re-compilations to create a complete test case). In any case, this means that all data are showing a very similar pattern and properties, and rightfully so – they are similar in nature as well. The coverage techniques follow a learning curve that clearly impacts the result when analyzing the timings for the different system in the coverage section.

10.1.4 Contribution

Our results indicate that the usage of a structural technique, coverage, has additional benefits for the quality of the test, compared to only using functional techniques. As part of our study we evaluated the efficiency and applicability of a set of TDTs: Positive/normal, Negative/abnormal, Random input, Equivalence partitioning, Boundary value analysis, Fault injection, and Statement, Branch and Method coverage. One of the major observations from our study is that our approach of a quick quality assessment and evaluation gives valuable information on the system quality. Through this pilot study we have also successfully experimented in our research approach for comparative evaluations of TDTs as well as established a framework which we plan to extend for future elaborate studies on complex/industrial systems.

10.2 Introduction

Our overall aim is to provide guidelines for testers in the software industry on selecting TDTs [65]. This can be achieved only through detailed studies on TDTs in terms of their efficiency, effectiveness and applicability.

Though our ultimate target is industrial systems, in this study we present a study of Open source systems and tools, with the motivation that the use of Open source is becoming a common practice also in the software industry. Consequently, there is an industrial need to efficiently assess the quality of these systems. We have explored our ideas in this pilot study – involving few testers and easily accessible and comprehensible systems – in order to gain necessary insights and experience to scale it to future studies envisaged for more complex/industrial environments. To establish the ability for the test cases to find failures and determine if they are contributing to coverage improvements, we have measured the effectiveness of the test cases. Testers are faced with the challenge to choose the TDTs that yield the most effective test cases. Testers seem to be biased towards choosing "positive" TDTs [194], even though these techniques are not certain to lead to really efficient and effective testing.

The TDTs considered in this study are:

- Positive/Normal input
- Negative/Abnormal input
- Random input (through an automatic tool)
- Equivalence partitioning/Equivalence class
- Boundary value analysis
- Fault injection
- Statement coverage
- Branch (Decision) Coverage
- Method Coverage

Our hypothesis is that the quality evaluation benefits from using both functional and structural techniques. It is common practice in industry to utilize coverage early in the test cycle (typically, in the testing performed by the developers themselves), but we have seen strong benefits also for testers to use such structural techniques. Many systems are often insufficiently tested, which implies a need for increased and systematic use of TDTs. We have measured the different systems in terms of how easy (fast) we can create and establish test cases, and at the same time assessed the time to find a suitable place in the software to use and apply the particular test case. Our assumption (based on Dijkstra [54]) is that from a testing point of view – it is easier to show the presence of failures, than the absence, which we also interpret as a guideline for selecting systems in the study. Since the quality of the selected systems is generally low, these

systems are also rewarding to test. Considering that these systems are all open source systems, one can assume that they have not been tested more than what is common practice for developers – which is clearly not enough for industrial use.

We will discuss our method and process after shortly presenting related work and terminology used. Subsequently we explain the TDTs, the systems used in our study and our results followed by a validity assessment, a discussion on the fulfillment of research goals, and finally our conclusions. Some additional details of our study are presented in [171].

10.2.1 Research Questions

The main research questions we try to answer are:

I. Is it possible to do a quality assessment of a software system based on limited test efforts?
II. What is the efficiency and applicability of the considered functional TDTs?
III. How can structural TDTs be used with functional TDTs to improve efficiency, effectiveness and applicability?

These main questions provide the foundation for a series of underlying questions that needs to be clarified:

1. How to characterize "limited test effort"? Is it restriction in time spent, restriction in number of test cases created, or restriction in coverage exercised? How much time does a test case take? How long can it take to understand where to apply it?
2. In what way does an increase in coverage improve the fault finding ability?
3. Is coverage a good complementary technique for a tester doing functional testing?
4. Can we trust the tools we use? Do different coverage test tools yield the same results for the same program?

10.2.2 Related Work

The literature on TDTs is vast, since many of these TDTs have been available for a long time. Major works in defining them are Myers [163]

and Beizer [19]. More recent work and discussions can be found in [203][157]. In Section 5, we present some of these techniques, and we have earlier defined a framework to evaluate TDTs [65], which is the foundation on which this work rests. The process described by Basili and Elbaum [13] is very straightforward and similar to the approach used in this study (described in Figure 10.1). A novel element in our study is the combination of metrics. We are not only testing the system, but also "testing our testers" addressing the questions: How well have they comprehended the TDT? and What are their experiences in using a specific technique on a system? Comprehension has been discussed as one parameter in Vegas's research [203] but not fully explored. Coverage is best overviewed in Zhu et al [220], but also defined in [29][93] [145][181]. Our plan is to expand and examine comprehension, as well as challenge the testers by giving common TDT names and approaches that are actually overlapping in many aspects, which gives us an indication of the data validity and the range of our results.

10.3 Process & Method Used

The process in Figure 10.1 presents the basic approach. The process is a simplification of our TDT evaluation framework, described in [65], and includes the following steps:

A. *The system selection*: The system selection will almost always have a profound impact on the result, as well as the people performing the study. We will discuss limitations based on this in the validation section, but a further background on why and how we selected the tools for this study is described in chapter 7. In this case we deliberately made a choice to investigate open source systems. Not only do these systems provide opportunity to freely publish the results, they are also a great target for creating test cases, since they often lack systematic test. The increasing industrial interest in open source systems also call for new approaches for efficient quality assessment of these systems.

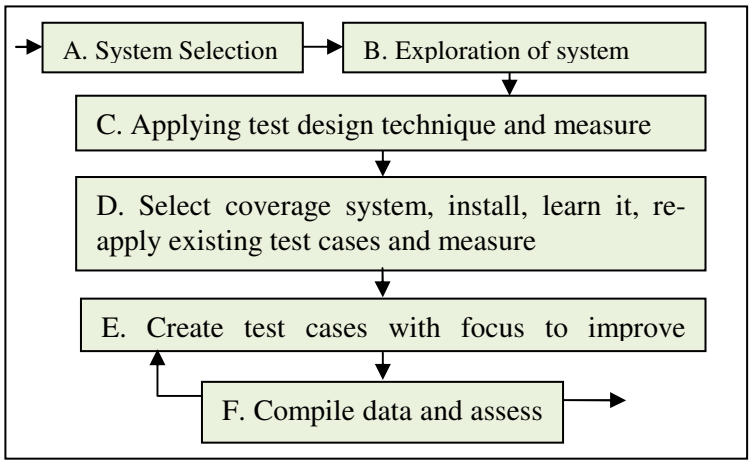


Figure 10.1 The process of data acquisition and analysis of this study

B. *Exploration of system:* This stage is necessary, since open source systems often lack clear written requirements or specifications, and documentation is insufficient. This gives a weak foundation for preparing test cases, which should contain a step by step description of execution and input, and must include a verdict to be able to evaluate the result of the test case (pass or fail). The latter (verdict) could for some systems be difficult to define, when nothing but testers personal judgments are used as basis. Correct verdicts are often formed based on the user interface, which is on the system level, and better verdicts are made if the tester has system domain know-how. This claim is our experience, where we also met situations where the opposite is true, meaning, our current assumption is that correctness of verdicts depends on the system domain. To be able to compensate for the lack of information needed to define the verdict, we have applied reverse engineering by exploring the system, and establishing what we assume to be correct behavior of the system, and thus identifying valid and invalid input and output data. After the pilot study of the two first systems, we added the measurement "time to explore the system". Then we spent time thoroughly analyzing the system architecture, to be able to establish the integration level of the system. After the fifth system, we re-assessed our approach. We dismissed the establishment of levels, concluding that the architectural picture gave little extra value, and that the

main levels for these small systems are the user interface representing the system and the source code. We also added the measurement time to first failure. After the study was concluded we decided to investigate coverage where that was possible. Due to tool availability, we could perform this only on a few systems.

C. *Applying the TDT and measure:* This means that we create test cases based on each TDT for each system and apply these test cases whilst measuring:

- Number of test cases for each test technique
- Number of failures found by execution of a certain technique
- Time to first failure
- Time to create the particular test case
- Time to identify a place in the system under test where to apply a particular test case for a specific technique

Not measured is defined as *n.m.* in the tables and *n.a.* information not available.

D. *Select Coverage system:* This step involves the complementary task of finding and selecting a code coverage measurement system suitable for the systems created in Java. Several systems were attempted, to find four that installed and usable, providing measurements, but with different usability features. The four systems selected are shown below. We also added information on installation time, and time it took to get the first coverage result when testing (handling the system):

1. Clover [39] (Installation in 15 min, first result in 10 min)
2. CodeproAnaytiX [41] (Installation in 10 min, first result in 10 min)
3. Codecover [40] (Installation in 15 min, first result in 76 min)
4. EclEmma [58](Installation in 45 minutes, first result in 90 min)

These four open source systems (freeware and shareware) are tools that can be used for measuring statement coverage, where particular 1 and 3 also provide features to measure branch coverage and 1 and 2 provided features for measuring method coverage. As a part of the selection, these tools had also to be usable on the different systems, so when installed and tried, only four systems were selected. This step of selection also excluded six other systems, based on different kinds of installation and runtime problems.

E. *Create test cases with focus to improve coverage, measure coverage:* This step is creating test cases by studying the code with the sole goal to improve the coverage. This also includes a learning curve which clearly shows in the results. By studying the code, the testers realized that several parts of the code were unreachable, and not an active part of the system, therefore code-adjustment was made to take away the code not in use (or for future use), and thereafter reapply and improve the test cases to achieve the best coverage possible.

F. *Compile data and Assess Result:* This step involves looking at the data collected, and analyzing and synthesizing the result, by calculating the following measurements:

- Failure/Test case (per TDT)
- Coverage and Failure Density
- Identify any systematic or significant pattern, e.g. which TDT has found most faults
- Assess quality and apply possible statistical methods

In addition to the process flow, validation is also discussed.

10.3.1 Description of TDTs and Related Work

In the plethora of definitions on different techniques, the hardest part is to describe the technique in a unanimous and unambiguous way. The span for different interpretation of these techniques is one of the factors that we will discuss further in Chapter 13, and in retrospect influenced our results. The TDTs used in these experiments are the following:

1. *Positive/Normal test cases* – executing what is expected to work according to specification, requirement, or documentation. This was mentioned by Myers [163] and used, but not clearly defined by King et al. [133][134] and more recently described in [217] as "…for the purposes of testing if a program does what it is supposed to do" or in [167] as "Positive test cases are aimed at verifying the fulfillment capability of an agent with regard to a given goal". Depending on the context and application, our target varies, but the core of the technique remains the same. Very interesting in this context is the positive bias in using this technique [194].

In systems with lack of clear specifications, this can be described as the "assumed" behavior of the system from a user perspective. Note that not only this refers to input techniques, but also to "clicking" or mouse movements (which in one way are inputs).

2. *Negative/Abnormal test cases* – aims to deliberately assess areas, expected not to be clearly specified in the system [217][142] or as described in [167] "negative test cases, on the other hand, are used to ensure an appropriate behavior of the system under test when it cannot achieve a given goal such as error management." This has been thoroughly explored under the name of "attacks" [211].

3. *Random input* [204], which in this case meant implementing a simple random input selection, given an ASCII-table to choose from. The simple random system is called The HAT [171], which basically defines a set of inputs (valid and invalid) to randomly choose from. Since the domain space of invalid data is often larger than the valid, the technique is biased against creating negative input. Random techniques are basically covering normal and negative input characters.

4. *Equivalence partitioning (EP) or equivalence class* [19][163] [127][171][182] is a technique that divides input values into sets, and the data in each set is assumed to be handled similarly by the system. In this study, there was an imposed limitation on this technique, which was restricted into being viewed as the set of integers.

5. *Boundary value analysis (BVA)* [127][161][163][182], assumes that input data is in an ordered set, that one can identify the boundary and thus create test cases for the adjacent values and the boundary itself. The goal is often to test the relations; < and > vs. <= and >=, according to specification. This technique would implicitly cover the EP, but adding at least one more test case (on the boundary). The result for each boundary is precisely three test cases, one at each side (adjacent) of the boundary and one on the boundary.

6. *Fault injection* is a method traceable back to at least 1967 [93] in the computer field, but used in other fields earlier. In particular this technique has been made common knowledge by Voas [204]. Fault Injection means deliberate change of a small part of the code (semantically, logically, or syntactically), and consequently one creates a test case that identifies the changed code as a malicious behavior of the system.

7. Statement Coverage [19][30][163][220] is a basic measurement initially intended to measure how much of the code lines have been exercised during test. Since instrumentation of line and statements differ slightly between different programming languages, statement is often defined slightly different.

8. Branch (decision) coverage [30][163][220] measures if each branch in a decision point is covered. Depending on the language the instrumentation is often different, and this is also often confused with decision coverage by different tools.

9. Method Coverage [220] measures how many of the methods that are executed (covered) in the Java program.

The techniques described above (1-6) were applied in the same order for every system, which was 1, 4, 5, 2, 3 and 6, and thereafter the three different coverage measurements 7, 8, 9 were collected in two subsequent trials.

10.3.2 Systems Used in this Study

Open Source systems have made software available in a new fashion. The most successful are operating systems, web-server support and database (GNU/Linux, Apache Web server, Mozilla, MySQL) [84][5][154][164], which have been the main drivers of the success. Open source is mainly "designer/developer" driven, with a focus on the code and component level of test [61][90][155]. Testers at function and system level are a must for any commercial and industrial software. These roles and levels are rare in open source community which means testing is also rare. System test is largely up to the user community. This poses the question; is it sufficient to test only with a code and developer focus? The answer according to the survey [90] is that the quality is higher from the developer's code level than for normal systems, but fundamental system test often fails to measure up to commercial products.

Table 10.1 contains some characteristics of the fifteen selected open source systems used in this study. Only system 3, 5, 8, 14 were used in the in-deep study of coverage. The table provides some maturity data, and the order they have been examined. Note that *n.a.* in this table means that data is not available.

These systems are from multiple domain types [171]:

- Student/learning (ID 1, 2)
- Personal Banking systems (ID 8, 9, 15)
- Image/paint and drawing systems (ID 5, 10, 14)
- Games (ID 11, 12)
- Management/calculate (ID 3, 4)
- Modeling tool (ID 6)

Many of the systems in this study can be found on either the Sourceforge or the PlanetSource community. We have selected rather small open source systems as examples, to fit the time-frame and scale of this study. Our systems should be small enough to be easily be installed and used on a limited PC, and our systems should not require long training to be able to understand the main goals, read requirements, help texts and other specifications. This also meant that the code should not exceed 15 000 LOC to make code review possible within the time-constraints of the experiment.

Table 10.1 Characteristics of Selected Systems

ID	System	1st Release	Ver-sion	Size	Down loads
1	JavaJusp [121]	n.a.	1	2Mb	n.a.
2	StudentHelper [193]	Aug'08	1	39Kb	30934
3	Clean Sheet [38]	May'05	4	1.1Mb	6631
4	Age Calculator [1]	Apr'02	1	16Kb	7208
5	ImageProcessing [113]	Feb'07	1	46Kb	5769
6	UMLet [199]	n.a.	9	6Mb	n.a.
7	IRC Client [114]	Dec'02	1	85Kb	31066
8	JMoney [122]	Mar'01	4	1Mb	3723
9	EuroBudget [74]	Aug'02	2	1.1Mb	7575
10	DraW [56]	Mar'08	2	342Kb	1404
11	Bomberman[28]	Aug'01	2	2Mb	n.a.
12	Bejeweled [22]	Sept'06	1	110Kb	6499
13	ImageJ [113]	n.a.	1	631Kb	n.a.
14	LaTeXDraw [141]	Jan'06	2	4.2Mb	n.a.
15	Bank System [10]	Oct'03	1	169Kb	33677

Finally, the systems should be interesting enough, to capture the interest of the people performing the tests, which in retrospect might have been the main selection criteria. No instructions were given to the testers to select systems based on age or maturity of the system. The first two systems lack some consistency in the measurements, and can be considered as the pilot for the systematic approach presented here.

10.4 Results of the Study

10.4.1 Time Measurements

Our first result, shown in Figure 10.2 is a measure of the average time to locate where to apply the TDT, i.e. the resulting specific test case location in the system, and the confidence interval (p= 0.05) for these times. The order is somewhat surprising. Random input is for many considered advanced and time-consuming, but once set up, this is a rather quick TDT to apply, since you only need to identify a place where "any type of input" can be given to the system. The time for random input test is including set-up time, and therefore significantly more time consuming than normal and negative test technique. If measured without the set-up time, this had changed the result. With p=0.01 (99% confidence) there is no significant difference between the applicability times except for fault injection.

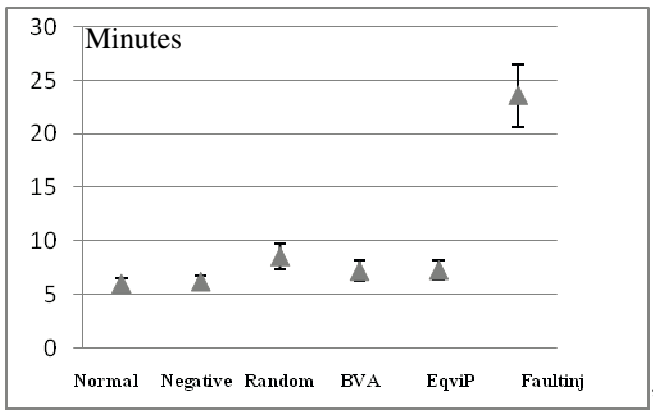


Figure 10.2 Average applicability time per test design technique and confidence with p=0.05

All techniques, except BVA and fault injection, possess this quality of allowing a wide variety of input and should have very similar times. It is noteworthy that BVA and EP juxtaposed to Random do not provide any significant statistical difference in regards to applicability time – and are also overlapping the times for normal and negative test, hence has no statistical significant difference at the 95% level. We noted that there was an ambition to create a "new" test case with each technique, instead of providing the same test case, but different input values, since the techniques subsumes (overlap) each other. That might have impacted the timings more. This impacts the data that it takes longer and longer time to find "a new input place", and order becomes more important.

Unfortunately we could not provide timing data for coverage in the same way, since the testers found it difficult to distinguish between the understanding of the source code (finding a location to apply the test case) and the actual construction of the test case, since for coverage, these actions are strongly interlinked. The data provides information on the data accuracy. It also stands out that Fault injection is significantly with 95% confidence more time consuming to identify and apply, which is expected, since creating the test case includes making sure the fault in the system propagates, or does not propagate (if that is the intention) to a failure visible in execution. Figure 10.3 describes how long time it takes to create a test case, once the place in the application where to apply it is located.

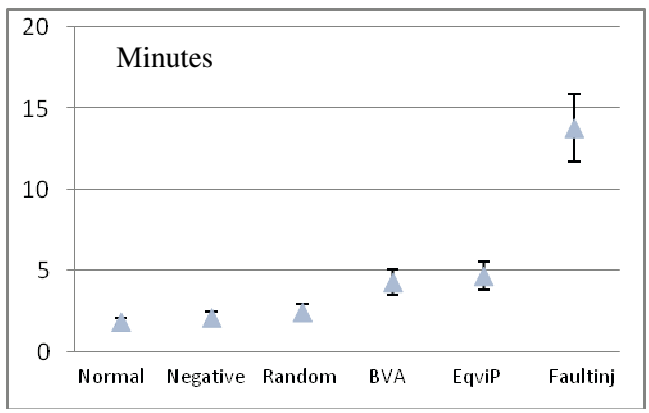


Figure 10.3 Time on average, to create a test case along with confidence interval p=0.05

Fault injection is a costly method – and significantly so even at 95% confidence. Part of this cost, gives a new perspective on fault injection, that this TDT is teaching the user how it is constructed, and is therefore a very valuable complement and asset in learning an unfamiliar system.

Once locating a boundary, and that means also, identifying what the boundary values are, the technique to write a test case is rather straightforward. The same reasoning goes for equivalence partitioning. One can conclude by the Figure 10.3 that on an average, the time to prepare a test case is short, and for most TDTs, the average time to find where to apply it is very short. This depends on the fact that few test cases are created (finding "the next" is therefore obviously easy), and that the systems are very simple to comprehend. The order of the TDTs could yield different timings.

The test cases here are different for each technique, but could have been the same. But since testers worked hard to test as much as possible, this option was not considered. If different data had been used for the same test case of all the overlapping techniques, this would probably reduce the time to create the test case, but not necessarily add to the quality assessment. In our experience from industrial systems, a test case can at its best take around 1 hour to create – but normal figures range between 4 hours to 24 hours. At a 95% confidence level, both BVA and EP has no statistically significant difference between them, but are significantly different than normal and negative test cases and surprisingly also than the random technique, and as earlier claimed, fault injection. The normal test case, could be only "clicking" around in the system – and thus not provide any input, since this is not a limitation that has been used for this TDT.

Table 10.2 shows for the systems under study (ID can be compared to Table 10.1) the actual number of test cases, the statement coverage, the time to the first failure, and the time the system was explored before testing started. The latter time influences the chance to "stumble across a failure", through execution. Unfortunately there are many entries with *n.m.* (not measured), due to several reasons in the approach. When looking for the initial time, and the "first time to failure", we cannot assume that these numbers are not influenced by the testers becoming more aware and efficient in their approach to the systems. One could say the "falling" times might be a reflection of the learning curve to focus on "where to look for problems". We noticed that the most difficult to test were the image and

drawing systems (5, 10, 14). In particular LaTeXDraw, since it did not allow wrongful input (one could not "draw" outside the boundary).

Table 10.2 Summary of the experimental data based on functional test (measuring impact in coverage)

ID	No of TC	Statement Coverage 1st time %	Time to First Failure	Time explored	Failure Density
1	18	n.m.	n.m.	n.m.	n.m.
2	30	n.m.	n.m.	n.m.	n.m.
3	21	49	n.m.	83	0.23
4	11	82	n.m.	12	6.05
5	19	n.m.	n.m.	63	n.m.
6	26	n.m.	2	40	n.m.
7	7	n.m.	8	35	n.m.
8	19	54	9.5	33	0.25
9	18	n.m.	17	37	n.m.
10	19	30	12	34	0.29
11	7	n.m.	12	30	n.m.
12	18	n.m.	n.m.	36	n.m.
13	17	n.m.	7	40	n.m.
14	14	13	No failures	35	0.00
15	19	76	82	33	0.79

10.4.2 Failure Finding – Effectiveness

In Figure 10.4 we can see which of the techniques (resulting in a test case) that found most failures. With a confidence intervall of 95% (represented by the line in the bar), we can see no significant difference in the effectiveness (fault finding ability) of the different techniques in this small sample.

One can observe that the most number of failures were found by the boundary value technique, which as a technique subsumes the following

TDTs: normal, negative and equivalence partitioning. One other explanation of this result is that BVA targets outside the boundary, hence negative test value. This indicates that these systems assume that the users are providing only "expected" types of values, which we claim is typical developer/designer behavior.

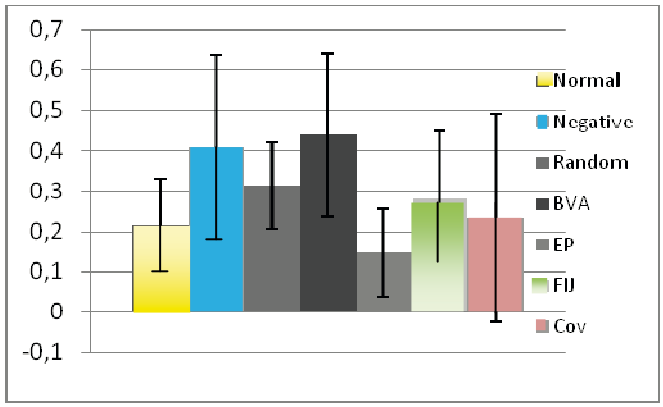


Figure 10.4. Number of failures found on average per test design technique, with p=0,05

Looking at Figure 10.4 again, if BVA did find most failures this would mean that this technique is very effective, when it is applicable. Normal, negative and random input techniques, as well as Fault injection, are always applicable techniques – where the main problem is that the data set is much too large. EP with the more generous definition would also fit into this reasoning, but has the advantage of doing some limitations of the data set.

Open source systems are often not well specified, probably as a consequence of little or limited written and documented requirements, and little requirements to specify the system clearly before or during development, since the specification will be the implementation. The result will be a larger "unspecified area" that is vulnerable to negative TDTs. Of the considered open source systems, 14 systems seem to have a large unspecified area, thus easily targeted by negative TDTs. The border is probably unclear in many systems. In large industrial and complex systems, this "unspecified area" will exist, mostly due to misunderstandings between

the vast number of people who are involved to cope with the size, which again motivates negative TDTs to be used.

The fault finding ability for the coverage technique is utterly intriguing. 39 new failures were found when using this technique, but the effort of 139 test cases to create and execute were rather time consuming. One pass for one system could take more than 5 hours to execute – since we did not anticipate automation of the test cases from the start. The only interesting data comes when looking back at Table 2, and concluding that the initial 17 test cases for LateXDraw (ID 14) only yielded 15% statement coverage (and no failures found), but that 17 test cases for JMoney (ID 8) yielded a starting statement coverage of 54% and as much as 29 failures found. Not until coverage was used, any failures ware recovered from LateXDraw, which could be an indication of the limitations of some functional TDT approaches for some systems.

10.4.3 Coverage Measurements

For the four coverage measurements, it is interesting to compare the time in Figure 10.5, where CleanSheet is obviously the first system of the batch, and the average time is much longer than for the other systems. Not having enough data for any proper analysis, the testers did confirm this finding, and they clearly felt that the more they learned about how to create the test cases to fulfill coverage measurements, the faster it went.

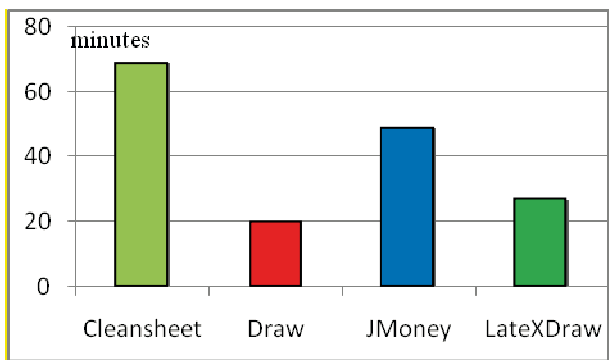


Figure 10.5. Time on average, to create and apply test cases, included test cases not adding to coverage improvements

What is a more interesting report is the fact that as much as 23 test cases (TC) for CleanSheet did not contribute to improved coverage, and was therefore discarded in the coverage measurement (but still counted for the timing). the same figures for discarded test cases was for Draw 13 TC, JMoney 11 TC, LateXDraw 21 TC, which also indicates the more trial and error approach of the technique, as well as the complexity of the code and time it takes to understand how it works. Our goal was to be a bit more exhaustive in our coverage testing of the four systems (ID 3, 8, 10, 14) we selected. Initially our aim was to achieve 100% statement coverage. We did a bad choice of not automating the test cases, which would probably have made the goal achievable. During our coverage aims, we found that as much as 19 functions were not available to use in the released version, and was in principle there "for future development". This can be considered dead code – and we aimed to have one final session where we removed this code. This was only true for two of the systems ID, and the reduction in e file size is under 5%, and as we will show – the coverage tools show such different results, that the change is omissible. Therefore all tests were performed with at least two tools.

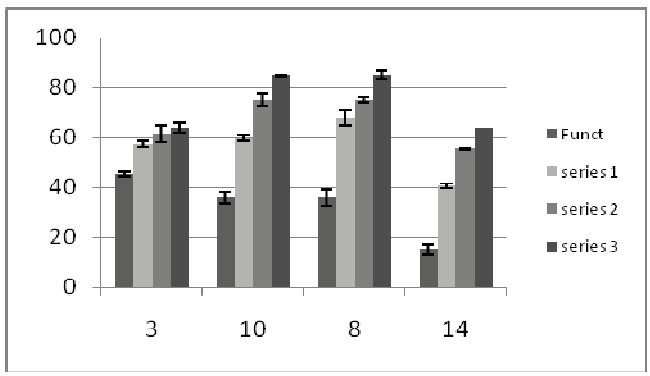


Figure 10.6 The statement coverage development. The confidence is p=0.05, based on n=4 for first two bars and based on n=2 for second two bars (n=coverage tools)

In Figure 10.6 we show the functional tests statement coverage as the bar on the left. From there the coverage growth can be visible, with the consecutive bars, from the three different trials. The first and second bars were measured with four different coverage tools, yielding possible confidence interval on the measurements at 95%.

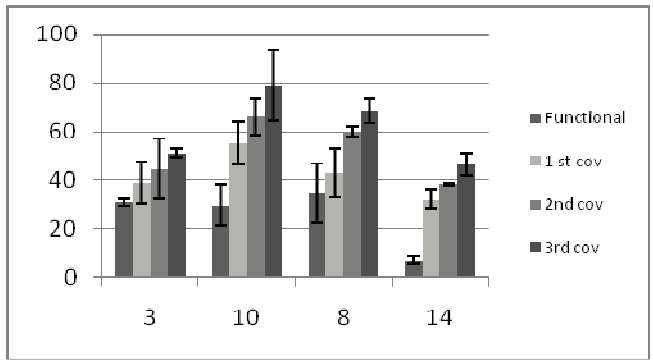


Figure 10.7 The branch coverage development the confidence is shown with p=0.05, based on n=2 (two measurement tools).

For branch coverage the improvement curve is visible in Figure 10.7. The first bar for each system represents the starting value (as in Figure 10.6) based on the functional tests. The confidence interval of 95% show a rather great uncertainty in the confidence due to the two different tools used showed very different values for the same test case execution. It also shows the difficulty to improve branch coverage, since very low values are achieved.

Finally the measurements for method coverage shown are visible in Figure 10.8. They show a similar pattern to Statement coverage and improve with each new iteration of the coverage-targeted test cases.

For the 15 open source systems in this study, we found that our effectiveness was around 32%, meaning that almost one out of three test cases yielded a failure, and the coverage has been between 29-82% for applying less than 21 functional test cases. Failure density is also rather high for the 4 measured systems. Even if both validity and significance of this result is limited, failure density is clearly below industrial and commercial systems. E.g. failure density is 0.25-6, compared to 0.00001 failures per KLOC in e.g. telecom systems (e.g. reliability requirement) [144]. The value of failure density related to all lines can be discussed, since it must be in relation to coverage (here we only use statement coverage) which are measurements that is often lacking, or computed differently than here.

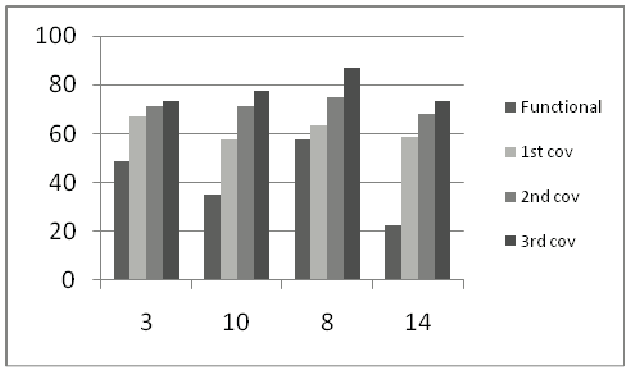


Figure 10.8 The method coverage development is comparable higher than e.g. branch coverage but shows similar patterns to method coverage.

10.5 Discussions

We have in this study attempted to answer the following initially posed questions:

I. Is it possible to do a quality assessment of a software system based on limited test efforts?

Since the results depend on the system selection method, which could "unconsciously" be biased to contain faulty systems and the testers knew this fact whilst performing this study, one could be hesitant to make any strong claim. But leaving this argument aside, we think it is possible to make a generic quality assessment based on these systems. If you found 27 failures from 52 test cases on a very small piece of software, none could claim that as good quality. Other supporting facts for our low quality assessment include a very short time of system usage and that the first time to failure was below one hour. We were using a very basic test approach and we could easily find faults in all systems.

The quality is undeniably low for 14 of the 15 systems, indicating that they are not sufficiently tested. At a first glance, only LateXDraw could be considered "ready for release" in a commercial aspect, but only 14 test cases might be considered too small of a sample, especially with the lack of adequate coverage data. Interesting is that the coverage data are also low for LateXDraw in the initial study, showing the difficulty to reach and

understand the system intuitively from the user interface. Not until coverage techniques were used, the first failure was found in that system.

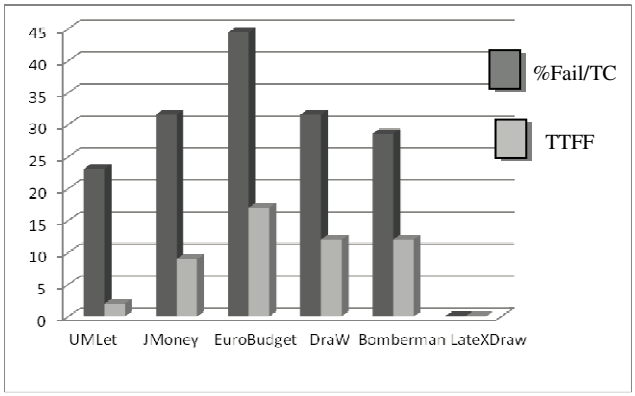


Figure 10.9 The % failure/functional test case and the time to first failure of the more frequently released system

II. How can structural TDTs be used with functional TDTs to improve efficiency, effectiveness and applicability?

From our results, there is no conclusive valid answer to this question. All TDTs are equally efficient, apart from fault injection. And this also goes for random, once the random is set up. The problem seem to be to find "yet another" area to locate where to apply the next test case, that causes time delays. Even if BVA and EP in Figure 10.4 looks like they are outperforming the other TDTs, (but there is no significant difference) it must be observed that the applicability *is* practically lower, since for two of the systems, no place for applying these techniques could be located at all. It is easy to understand why BVA is not applicable, since it requires an ordered set of numbers – sometimes not available in the user interface, but that EP was deemed as not applicable is due to the fact that EP was defined as only working for the same ordered set of integer numbers in this experiment. A useful and thus more generic definition makes the technique applicable on any type or set of inputs (by grouping classes of input, and assuming the software treats the input data in the group, in a similar fashion). Since the more narrow definition of EP is used this also impacts the applicability of the technique. Test cases targeting negative test cases and outside the boundary (negative, random, EP and BVA) all finds

failures. The fact that all types of techniques find failures indicates that the quality in general is poor.

III. How can structural TDTs be used with functional TDTs to improve efficiency, effectiveness and applicability?

Coverage seems to be a valuable complement to create test cases finding additional faults. It is hard to compare coverage and functional tests, since exhaustive attempts on the functional tests were not applied before attempting the coverage. Therefore we view our results with a specific target to achieve a test goal with limited time that coverage seems to be efficient, when understood. Furthermore coverage did reveal failures for LateXDraw, which functional tests failed. Part of the reason might be that the possible aspect of application was not fully understood, without looking at the code. This also indicates that only test through the interface without using any coverage could be very poor testing.

IV. How to characterize "limited test effort"? Is it a restriction in time spent, restriction in number of test cases created, or restriction in coverage exercised? How much time does a test case take? How long can it take to understand where to apply it?

We approach this by setting restrictive parameters, like not spending more than one hour playing with the system, measuring time to locate where in the application the test case could be constructed for, and then measuring the time it took to create the test case. For the tool dependent coverage, we realized the process of reading code and learning the tool must also be a part of the applicability. We can conclude that the testers experienced a strong learning curve which is somewhat visible in the data, where Cleansheet as the first system has longer times (see Figure 10.5). The restriction was further visible for the functional systems, that by all means were not exhaustive, but a few test cases in each TDT were attempted. The coverage measurement was finished based on the long execution times of all the test cases that in the end were easily over to 5 hours. This results in the conclusion that automation is a necessity to perform efficient experiments with coverage. We were glad to conclude that the functional test cases were fast to create, but we failed to measure them in the sense that "the next" test case would take longer. We assume that this would be the question for BVA and a restricted definition of EP. Fault injection was substantially more costly in time.

V. In what way does an increase in coverage improve the fault finding ability?

Statement Code Coverage and size are strongly related, so it is not surprising that the smallest system also achieve the highest coverage with 82% for only 11 test cases. By applying coverage consciously, more features could be discovered, and also more failures. By increasing coverage from around 40 % to around 60 % we found 19 more failures. This was a guidance based on the fact of adding 31 more test cases. Even if our sample is too small to be conclusive this fuels us to in the future to add test cases to achieve better coverage, and measure how many more test cases one need to achieve 100% coverage. Or aim to compare how "exhaustive" functional testing would work for a very small system (and at the same time monitor the coverage growth) in competition with a pure coverage approach. With our 167 added coverage test cases, we did found additional 39 failures in all systems. The data is not conclusive, but it gives a strong indication that the approach is feasible.

VI. Is coverage a good complementary technique for a tester doing functional testing?

We do not possess conclusive evidence for a claim, but it seems positive enough to investigate for further studies. A fact is, that not until we applied coverage to one of the systems, we could unveil some failures, which indicates that coverage is a promising complementary approach. Hence, low coverage was achieved by the testers only using functional testing through the interface, (e.g. black box testing), testing targeting coverage could provide information to challenge the software better. If coverage is performed before the functional test the situation might be different, and this aspects must be better explored in the future.

VII. *Can we trust the tools we use? Do different coverage test tools yield the same results for the same program?*

We have shown that coverage tools should not be trusted as fact, and their instrumentation will have the actual result (measurement) vary a great deal. This added an extra interest in our measurements and their trustworthiness. Our experience with this tells us, that this is an important note to organizations claiming certain coverage result, or demanding them in business deals, but also in research accuracy. We have shown that coverage results are not trustworthy since they depend on the tools instrumentation, resulting in different coverage percentages. The standard deviation for

statement coverage was 3.6 %, and the measurement distance of 8.1%. For Branch coverage this was even a greater difference, whereas standard deviation is 12.5 % and the measurement difference is as much as 17.7%. The conclusion is that Clover was the most appreciated test tool used, since it handled all measurements required for the task, and also could show where in the code the improvement existed.

10.6 Conclusions and Lessons Learned

In a study of 15 open source system we have gathered information on the quality of the systems under test and found important failures that show that it is possible with only a few targeted test cases and very limited time, to yield valuable information and a reasonable coverage. This study has by its approach added another meta-level of evaluation, where efficiency and applicability of such techniques were in focus.

Our conclusion is that only one of the fifteen systems seemed at a first glance well specified, since "by design" this system have limited the possibility to give invalid input, making the system less vulnerable to failures and also making negative test approaches non- applicable. Not until approaching the system with coverage, the testers found uncharted areas that were then possible to create test cases for and locate failures. We also conclude that all TDTs yielded test cases that found failures, which consequently means that quality was rather poor for most systems.

Even if both validity and significance of this result is limited, failure density is clearly lower in industrial and commercial systems. E.g. failure density is 0.25-6, compared to 0.00001 failures per KLOC in e.g. telecom systems. The only difference is how well specified and how much tested the system is upon release. Even if fault injection takes much longer time and can be deemed as not cost efficient, it provides a new way to obtain a better understanding of the system.

We found that coverage testing has a clear learning curve, when the initial approach might feel time consuming, but once the source code is more familiar, and the tools are appropriately showing exactly what code row that has been passed, the task gets more efficient. We were also pleasantly surprised by the low cost and the high yield of the random input testing, which should ignite further studies. And – in general – which of the techniques that are used is arbitrary, when it comes to cost of construction

(excluding the more expensive Fault injection). This is good news, since we can then focus on the techniques providing test cases that finds faults, are applicable, and are easily understood by the tester.

The TDTs that find most faults are without doubt the negative TDTs, or techniques that include invalid input. This implies that the behavior or use of the system is often specified, but it is not defined "what should not" work. This also indicates that the users are typically assumed to be familiar and to "know" what the right behavior and input is for the system, something that cannot be taken for granted, especially not in open systems.

Chapter 11. Systematic Mistakes in Test Case Construction

11.1 Summary

Test Design – how test specifications and test cases are created – inherently determines the success of testing. However, TDTs are not always properly applied, leading to poor testing.

We have developed an analysis method based on identifying mistakes made when designing the test cases. Using an extended test case template and an expert review, the method provides a systematic categorization of mistakes in the test design. The detailed categorization of mistakes provides a basis for improvement of the Test Case Design, resulting in better tests. In developing our method we have investigated over 500 test cases created by novice testers. In a comparison with industrial test cases we could confirm that many of these mistake categories remain relevant also in an industrial context.

Our contribution is a new method to improve the effectiveness of test case construction through proper application of TDTs, leading to an improved coverage without loss of efficiency.

11.1.1 Context of this Study

Our main research questions in this study is “if there are systematic mistakes testers do frequently during test case construction”, which leads to reduced efficiency and effectiveness of the testing effort. This study was a result of looking at the students’ competence in applying the TDTs. This study is the result from the third investigation on Students, as we earlier discussed in Chapter 7.

11.1.2 Design - Research Method

We have collected empirical data during test case creation, formulated a theory of systematic mistakes, and compared our theory on existing test cases written in industry. Our aim is to define distinguishing features of the test case design process that can be applied generically to different types and domains of systems.

The primary goal was to teach the students how to write test cases in practice, transforming their theoretical know-how into useful test cases. There were about 50 students participating in this study, with the target of creating 10 test cases each. Not all students delivered, and not all test cases were written. The students can be considered novices in testing real software. The students were then asked to do a rather controlled exercise to apply their theoretical knowledge. The same system, Buddi [32], was used. Since it is easy to understand the basic functions of such a system, no requirement specification was available to the students, who had to use their own judgment to create reasonable test cases using different TDTs. Each student was asked to use a series of TDTs and fill in a template. The template was explained, and clarifications provided whenever necessary. Students were particularly asked to be innovative – providing new and novel test cases – something that would also give them extra credits.

11.1.3 Validity Threats

In regards to conclusion validity the major threat is that the collection and judgment of data poses some researcher bias and the categorization might have become different if another person would have qualified and decided on some of the tricky borderline cases.

The main potential threat to internal validity is diffusion or imitation, since respondents could have been influenced by each other. There was no way to check this, since all had the same system under test. This means, that a result with many mistakes could have been spread among students as correct, and thus negatively influenced the result. The interviews at industry were rather informal in nature, and the researcher could have influenced the result.

We conclude that the threat to the construct validity to be limited, since we have explicitly measured their frequencies, based on our definitions of

mistakes. The evolutionary nature of this study and the fact that the original intent of this study was different, could pose as a threat to the construct validity.

The major threats to external validity, answering if these results are possible to generalize, can be discussed. It seems like our result is just a first attempt, and that replicating this result is an obvious next step. The results from the TDTs in *Figure 11.2* are not possible to generalize since they are dependent on the system under test.

Another confounding factor not taken into consideration is that the student group and industry are from a limited selection. We believe the amount of experimental data is appropriate and adequate for such an initial study. However, we have no sufficient data to support the elaboration on what consequences this has in industry, since it was a convenience sample. Therefore the result for the new and added category identified in industry is not sufficiently supported as a systematic mistake. The industrial trial must definitely be better randomized, using more respondents from different industries, so that the view can be generalized.

11.1.4 Contribution

Our claim is that a deeper understanding of mistakes that are made during test case construction, and conscious and directed efforts in avoiding them, will lead to substantially better test cases.

The strength of our proposal lies in assessing the likelihood of making a series of mistakes, the penetration of mistakes made, and the identification of consequences thereof – which all are aspects that enable a better defined test design process, resulting in improved test cases.

11.2 Introduction

In academia, a TDT is assumed to be known when published, and thus the implementation or interpretation of a technique is inherently assumed to be correct. Due to the vast number of publications and existing interpretations, in addition to a large overlap of the different TDTs, this body of knowledge is far from being well-defined enough to be an unambiguous source for use by the practitioners. From an industrial perspective, test design is

paramount since the quality of the test cases substantially affects how well the system is tested, what failures (faults) will be found and what coverage can be achieved. Our main research questions in this study is "if there are systematic mistakes testers do frequently during test case construction", which leads to reduced efficiency and effectiveness of the testing efforts. By *systematic* we mean repeating and frequent pattern occurring for more than 10 persons and more than 50 test cases in the context of the study. In an industrial setting the number should probably be lower, e.g. 5 persons.

11.2.1 Overview of Test Design

Test design describes the phase in a process, where test specifications are written, and a resulting test procedure or test cases are created. Following IEEE Standard 829 [110] (1998 version) a *Test Design Specification* should consist of:

- Test Case Specification Identifier
- Test Items; *(references for traceability)*
- Input specifications & Output specifications;
- Environmental needs;
- Special procedural requirements;
- Inter-case dependencies.

This specification is straightforward, but does not include the bookkeeping information normally used in industry, such as information about version handling.

A *test case* is the result of applying a test (design) technique to a specific software system. The test technique delimits the type of test cases that can be created, according to a concept, approach or selection. In industry, there is typically no extra level of documentation for the test procedure. The test case includes all needed information and is the executable similar to the test procedure, regardless if the test case should be executed manually or by a tool.

A *test case* should be *repeatable* by anyone (yielding the same result) and thus measurable, in the sense that it should be possible to determine if the test passed or failed. The test procedure in the standard also calls for a wrap-up that describes the actions necessary to restore the environment (referred to as clean-up in our study). The related work deals mainly with

software faults/ failures and making improvements on how to avoid them [63][66]. Improvement models as a general modus operandi is found in e.g. [138][33]. Particular work on improving the test design phase and assessing the test cases in this manner is, to our knowledge, new.

11.2.2 Origin and Design of Study

Our overall aim is to improve the industrial testing practice. We are focusing on the test design phase, and on the efficiency, effectiveness and applicability of the techniques used in this phase. The process of data collection for this study is described in Figure 11.1. The first phases, I and II, were set up for another study (hence are shown by dashed boxes), where the goal was to understand the know-how in industry about TDTs and their usage.

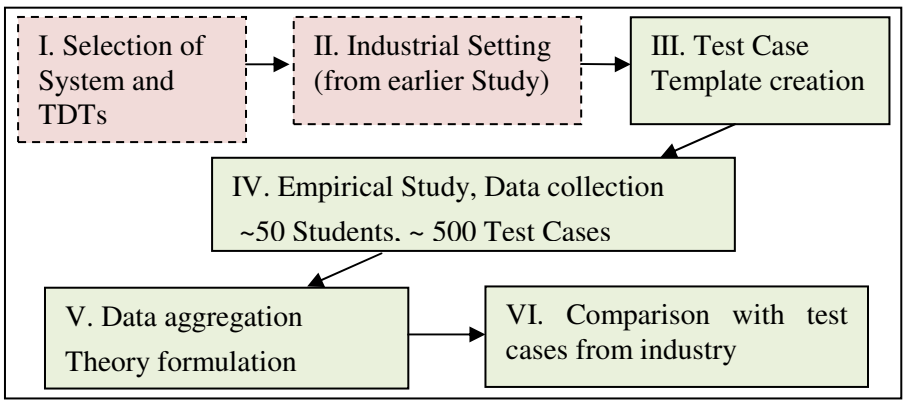


Figure 11.1 Process of the Study

A simple open source system, Buddi [32] was used, since it was intuitive to learn and small in size with a limited number of functions. The system handles a personal budget, creating accounts, making budgets, handling cash withdrawal and deposits, etc. and is very rudimentary – handling all input as strings, except some numerical fields.

We believe that any system with a set of test cases can be used to replicate this study, as long as the know-how of the system does not in itself become a hurdle – which will otherwise make the experiment unmanageably time-consuming. In phase II, all subjects were from industry (both testers and

developers), including some with more than 30 years experience. As a part of this study, the subjects were asked to write test cases on a blank study (since we assumed their experience in writing test cases would be sufficient).

Observations from this study include that the quality of the described test cases was in general very poor, even if our research questions were answered. We saw this as a result of the subjects being time constrained, but noticed that the test case writers were very brief in their descriptions. We also observed that since no template was given, the variation of detail in the produced test cases was large – a few wrote a much detailed step-by step description whereas most test cases were written rather schematically. These observations left us wondering, whether the underlying problem was the inherent complexity of specific TDTs, or if it was due to a lack of know-how of the TDTs, or if it was really just a case of poor test case writing. To identify and analyze the main causes of the observed poor test designs, we decided to conduct a large-scale experimental study. However, since we were unsure about how much the knowledge of industrial testers will affect the results of such study. Due to practical reasons, we decided to perform this study focusing on novice testers in an academic setting. As our study turned out, studying mistakes patterns are much easier if they mistakes are made frequently. This main empirical study was conducted as an element of a Master-level testing course at Mälardalen University. In preparing this study (Phase IV), which focused on the understanding of TDTs and the ability to apply them, we used the lessons learned from phase II that in order to get properly documented test cases a test case template is needed (phase III in Figure 11.1).

11.2.3 Test Case Template

In this study we used a test case template based on IEEE Standard 829 [110] with some additional fields (marked by *). The template contained the following fields:

- Test case name (& number)
- Test suite, (version)
- Test technique used *
- Time to create the test case *
- Version or unique reference to:

  - test items (test object lists, test artifacts, test plans etc).
  - software under test
  - project & product
  - test tool
  - test environment (configuration)
  - test specification (version)
  - requirement (version)
- Assumptions (pre-requisites) *
- Starting position of test case (implicit, the inter-dependencies)
- Input specification (input analysis, and selected targeted input)
- Step-by-step description of actions (procedure) of actual test case
- Output specification (observable outcome to base evaluation on)
- Clean-up including side-effects (post processing) after test case execution

The test case template along with the related experimental data for this study is available in [194]. The aim was to create industrial-like test cases, and the template helped the students describe the test cases with high readability. Time to execute (not create) the template is used in industrial test cases, but since we used trivial test cases for the students, we thought the creation time would be more interesting. Most students ignored or defaulted this category instead of measuring it. For the field “assumption”, the tester was asked to provide information on what (s)he believed to be the relevant system response to the test case, since there were no requirements or other specifications available for the system under test. This field was intended to provide information that enables definition of a suitable verdict/result.

In an implementation, the starting point of execution is particularly interesting – but many testers code their test cases in a particular order, assuming that this order is followed during test execution. This creates dependence between test cases, which is undesirable since a test case should always be self-contained. Defining intercase dependencies is a requirement in the standard, but instead we required the specification of a starting position (which indirectly indicates if the test case is dependent – or self-contained).

11.2.4 Considered TDTs

In this study, the following TDTs were explicitly taught in theory, including some simple examples:

- The positive test case (valid input data) (Pos T) [133][134][163][217]
- The negative test case (invalid input data) performed twice (Neg1 + 2) [142][167] [211][217]
- Magic input test case (0, -, float or other typical fault invoking data) [19][146]
- Equivalence partitioning technique (EP [163][127] also referred to as Category Partitioning [171]
- Boundary Value Analysis (BVA), [19][163][142]
- State-transition – preparing the model for the test case (based on the system) (STModel) [217]
- Use State-transition, and make the transition in 3 steps in the test case. (Steps can be transformed into a table.) (STable)
- Permutation of transitions/steps (identifying a location where that is possible!) [49]
- Combination techniques: State-transition + input analysis (Add EP-classes) (Comb)

11.3 Systematic Mistakes Analysis

The data collected in the above study was aggregated, and treated statistically, as indicated in Figure 11.1 (phase V). A bit surprised by the rather large amount of test cases lacking a sufficient level of quality, we then tried to identify what had gone wrong. After an iterative process of identification, grouping and refinement, some patterns that seems to posses the same qualities emerged. Based on our findings we formulated our theory on systematic mistakes presented in this section.

We had to define a way to determine the quality of each test case as a matter of grading. We noticed that the students had a strong tendency to repeat mistakes. If they did miss one category, they probably did that for most of the test cases. Then we could a see pattern among many of the students, on why they failed. After this analysis the structure emerged as the following list of categories, each indicating a lack of understanding why the corresponding knowledge is important for test case design:

A. Understanding instructions /level of details
B. Understand the purpose of the system and current level and context of testing
C. Understanding TDTs and how to apply them
D. Assumptions, e.g. regarding correctness and completeness of specifications
E. Elaborate test case creation, and not only using the most obvious test case or input
F. Define a clear starting position for the test case
G. Make specifications of valid and invalid inputs
H. Step-by-Step description of test case execution
I. Test case evaluation (steps to take to make a clear comparison with expected result should be clear)
J. Clean-up after test case, repeatability

In the following subsections, we will for each of these categories, present the data supporting our claims, the degree of failure for novice testers, and discuss some of the consequences of failure. In Section V we further relate our findings to the quality of a set of industrial test cases and enhance this categorization.

11.3.1 Understanding Instruction/Level of Detail

In test design, frequently imperfect specifications need to be translated to useful test cases. This process is the same as implementing a code or design based on requirements. In complex systems, the information in the requirements is often insufficient, and additional information must be gathered from different sources, but also be drawn from the testers or developers experience on how the system and software behaves. A tester should be able to infer exactly what is meant, and in a detailed level follow instructions and provide enough information, so that the test case is unambiguous and can be repeated by any other tester.

Observations: 33% of the subjects did not read instruction on delivery of test case (naming, or delivery faulty) and 73% did not complete the entire template.

Discussion: Not being able to read an instruction is in itself a failure, which could have many reasons. Here we assume that the students were

either not interested or did not think it would matter, or just did not care. Examples of mistakes are:

- Delivering the test cases in one file instead of as separate test cases uniquely named
- Wrongly name the test cases against instruction
- Not filling in information as required by instruction

If it had been a recruitment situation, people would probably not get a job as a tester based solely on the lack of attention to detail. In general, failing to provide detailed descriptions seems to be a human fallacy – where we in general are much too imprecise to make coding and testing a straightforward matter. Often the main solution is to provide more details – and by doing so, minimize the opportunity for multiple interpretations. The consequences here are often failures based on misunderstandings, or that the task of test case writing cannot be completed due to insufficient information. We can see that the imprecise level of detail is often generic in many of the categories used. The problem was systematic to a person, and not very varying with each test case, but we could also see deterioration in the test cases and also a strong relation to the success of applying the technique.

11.3.2 Understanding the Purpose of the System and Current Level and Context of Testing

Understanding the purpose of the system, and current level and context of the system is related to the abstraction levels of the system. This is probably the most fuzzy and hard to grasp concept of a software system when it comes to testing and seems to be an understanding that people acquire after some years working with the system. The system impact could be based on the history of the system, for instance how well documented the original requirements are, and how the history of the test has been. Who has been writing the test cases? What level were they?

This impacts how data is stored and handled, and also how the test case construction looks like. Is a test case written directly in code – or is it textual and manually executed? Is the test case hidden in a tool, a model or are there many documents and specifications to be read about what is expected? How do you actually learn about the system? Are there many similar systems on the market? Are you as a tester also a typical user – or is

the system where you test a constructed artificial interface? System impact is important in many aspects for understanding levels and context e.g. what visibility of the domain is possible, what software concept is used, and how that does impact the test approach.

Observations: Measuring this comprehension is rather difficult. We checked how many had understood that all input in the system were string-based, and did not create test cases that would assume the test should only handle digits and letters. As much as 80% of the students failed on this account, which led to a majority of test cases failed.

Discussion: By failing to understand the right context of the system, the likely outcomes are that the important test cases are missed, the focus of the test is outside the scope or at the wrong abstraction level of the system, and that the problems reported might be unimportant.

11.3.3 Understanding TDTs and How to Apply Them

One can discuss the TDTs and details, overlap and variants, in depth. The first and obvious level is to understand what the theory is, and then you need to be able to apply the technique on the specific case in your system, meaning, finding a location or situation where you can apply the technique. Most TDTs are related to input, some are related to path of execution, and few are related to order of execution. Also combinations of techniques are possible. Depending on how, and sometimes what type of system you apply it to, they carry different names. When looking at variable input given, the best way to get a good utilization of TDTs is to define the input domain and divide it into groups and subgroups. A good basic approach to have in mind is that the entire ASCII-table should be taken into account and to that a variety of different sizes should be submitted. This basic approach is normally documented in the test specification, and should cover all forms of valid and invalid input, whereas the specific test case should select a specific executable.

Observations: The success of using various TDTs by the students is shown in *Figure 11.2* (abbreviations are defined in Section III, D). The positive test case technique (e.g. giving valid input), was the most common technique performed, and also had the highest success in producing an

executable test case. Only 10 % did honor that for BVA all three data must be executed, even if this was highlighted during classroom teaching.

Discussion: The most common mistake seems to be related to understanding what is required for each technique, and also what input and boundaries really mean in the context of this system. One mistake is that students confused negative (invalid) input to using negative numbers. Another mistake is to confuse boundary values to negative test cases, specifically, input outside the boundary. In reality, there are probably many more subgroups, but this needs further analysis. The main goal is often to create a test case for each input-class or sub-group. One can define each input class or sub-group assuming that the software treats the input equally within the class (but it might not be true!).

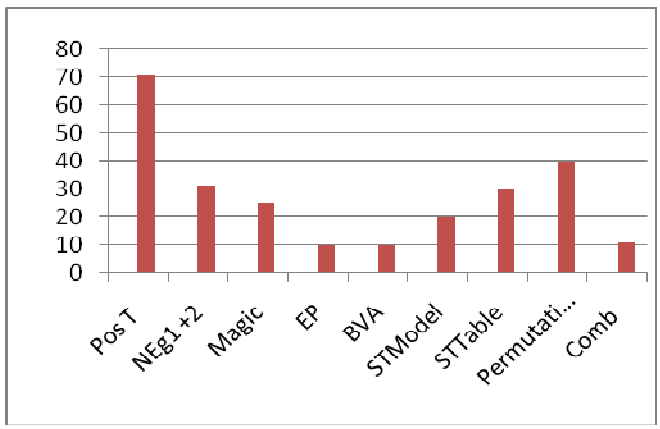


Figure 11.2 TDTs % Full and partial success in constructing executable test cases in the technique

A consequence of this is that there might be a series of variants of test cases, where the input varies (and accordingly its output), and as a result there will be a number of variants of the same test cases existing. In addition, if a boundary exists, it is valuable to target that immediately (three test cases – or one with three types of input). Boundaries are more or less visible at different levels in the software system, but should always be a target to test – since it is a known source of problems.

When using TDTs to create test cases, the first aim should be to create test cases to gain as much coverage [221] as possible, by e.g. varying the input. This has a higher likelihood of finding faults. Another technique,

permutation, suggests changing the order of execution of different test cases, or changing order between steps within a test case. Permutation mainly targets resource dependent faults. Finally, the context could differ for the same execution – for example if different security roles exist; or depending on whether other parts of the software are present or not. This means the same test case can have totally different execution paths and results, depending largely on the context of the execution of that test case.

11.3.4 Assumptions

This category relates to on which assumptions we judge that a test case has passed or failed. An experienced tester is likely to make judgments on correctness and completeness of all aspects of the system. In the case of judging a faulty requirement, experts would be more inclined to assume that the requirement is incorrect and should be changed. The novice would assume that what is written is almost always correct, and design the test case based on this faulty assumption. In this case, most students did not accept the system, and defined assumptions outside its current behavior.

Observations: More than 50% of students failed to create an assumption that matched their expected result. Since almost all students failed to identify what to expect based on an understanding of the system, they failed making realistic assumptions in relation to that.

Discussion: In this case, inventing an assumption became totally unrealistic. We decided this was more a consequence of not understanding the particular context of the system, rather than making a fault assumption. In fact, here we were more interested in understanding how the student could postulate a defined truth – what should be valid about the system, since no requirements or documents existed to explain what would considered being a correct system behavior.

11.3.5 Only Using the Most Obvious Test Case or Input

During analysis of the several hundreds of test cases, we were interested in seeing the variety of test cases created. We particularly asked the students to be creative in inventing valid test cases for the system.

Observations: Only a few individuals (less than 5%) had any variation within the system, or attempted anything innovative with their test cases. The student most often tested the function create account and the variation was very limited (mostly names and numbers were tested). The second most common test was looking at dates. A few did attempt to look at some transactions which led to more meaningful tests with slightly higher coverage.

Discussion: This mistake will result in systems where the obvious aspects are probably the only parts of the software which are tested, leaving many aspects of the software untested, and leading to less robust systems. In fact, this is an ineffective way of testing software. Doing a transaction test, will additionally test that the accounts must be created and can be used. This is a much higher level of test approach, than checking that the software can store a date, which in this case was a string. This type of tests had little impact on the main functions of the system.

11.3.6 Starting Position for the Test Case

A common mistake is to only describe where the starting position of the test case is, i.e., not being specific on how to get to the starting position or which actions has to be taken before; or just assuming that a particular location is obvious from the test case context, or not saying anything at all.

Observations: 73% of the students failed on explaining an unambiguous starting position.

Discussion: This would of course be easily detected during automation of the test case that information is insufficient to identify where the test case should start. The most common assumption for this exercise, which targets test of a very small program, is assuming that information such as “Start the program” is enough. In this case (and since we used an old version of the software), one has to avoid downloading the new version automatically, which none of the students remarked in their starting position. Secondly, one could benefit from describing exactly what to invoke and what part of the software state should be available. A starting position defines if the test case is independent or has intercase dependencies. A consequence if you do not create restart options for your software is that if a failure happens, the execution might get stuck, and will not be able to continue on to the next test case. The investment done in the test cases are better catered for if the

order of the test cases can be swapped around, especially if the software execution is in any way handling or impacted by resource factors in the system, e.g. timing, buffer-sizes, priority queues, caching etc. The aim is of course still to minimize the overlap in repeating starting positions, and thus prepare a set of starting positions. This is a part of the test architecture that is needed in the testware.

11.3.7 Specification of Valid and Invalid Inputs

Defining input classes in the test/verification specification simplifies the use of different TDTs, and the input selection of the test case. This is an efficient way to capture the entire input domain, and also prepare for a series of TDTs. At the same time, one could introduce variables to represent inputs, thereby paving the way for test automation. It might be initially difficult to define what an input is in this context, since clicking on a predefined menu-item could at another abstraction-level be regarded as input. Here we define input as something you enter in a field that can vary the execution path. We are particularly excluding pre-defined clicks or selections of menus (or commands). The best option for this type of user input is e.g. the ASCII- table, including digits and letters in addition to other special characters. In addition, a varied size of the input (defined in range) should be explored.

Observation: This proved to be very difficult for the students, and only 30 % of the students were even near the idea intended with input analysis. The students were in general better at providing valid input than making an analysis on invalid input. None succeeded to grasp the entire input domain.

Discussion: The consequences of poor input analysis are several. First, the test case created becomes ineffective, since it cannot be reused with many different inputs that would increase the coverage. Secondly, the analysis enables better usage of TDTs and thus better automation of the test case execution. This is done using input as a variable saving serious space, instead of hard-coding input data. In the context of the experiment, the problem might be how this concept was taught and clearly theory alone was not sufficient.

11.3.8 Step-by Step Description

A test case procedure is often described as a set of actions, with a step-by-step description of actions (and maybe also intermediate responses). To make the execution path uniquely defined, the test case must often be described in small and very detailed steps, exactly the same way as writing code. Otherwise pseudo-code could be an intermediate step, but one can rather question if working with testing software should be done by people with no knowledge of software or coding. The task would be to make clear what information is needed in the code procedure or step-by-step description – and what are comments, or header/book-keeping information about the code. The latter is an absolute necessity to be able to handle the test case.

Observations: 47 % of the students provided insufficient information and detail to provide steps that could be unambiguously followed by another human, or required mind-leaps that is needed to be added when creating a program for execution.

Discussion: We could see that beginners (not thinking in code) are writing too little information in the test case. Most common mistake is missing or too abbreviated information, which probably makes the automation of a manual test case costly.

The consequence of missing specific detail in the step, e.g. what particular values that should be used, is that the test case execution path is not uniquely identifiable and the execution might not be possible to recreate, thus if a fault is found it could be hard to recreate the test case.

11.3.9 Test Case Evaluation

The main purpose of test execution is to get a measurement of the software quality, by combining a large series of test case evaluation results. To be a useful test case, it must be possible to evaluate the outcome of the test case to the defined criteria. For systems lacking these criteria the concept of "Assumption" is useful. To determine the verdict of the test case one must describe the expected result or visible outcome, so that the outcome of the test case could be compared with it. This could result in a series of steps, comparing logs, or showing that certain action took place.

Observations: As many as 28% missed giving evaluation at all, and about 50% of the students could not formulate a precise evaluation that would determine the outcome.

Discussion: In our system, when an account is created, it is not enough to make sure that the test execution did not encounter a crash, fault or problem when pressing save after filling in an account name. One must also perform the action of retrieving the account name into a visible state, e.g. create a listing of account names. This also means that the test case for checking that listing must be working. Another way is to go through the back door, and check in the data base that there is storage in the correct table with the saved name. Both must be precisely described. These sorts of events create problems when designing and testing the system. If the name is not visible in the list, is it the list function or the store function that is malfunctioning? In systems there are often multitudes of ways to check a specific aspect. In our system, one can try and repeat exactly the same test case immediately. The next time the test case should fail, since it probably would not be possible to store another account with the same name (assumption). Evaluations are in some systems the trickiest parts. Observation is low, and one has to either wire-tap that the information or signals really passed – or do some complicated analysis to form a judgment. If this is left out, the testers are at loss - but also – the testing is not complete. The best questions to ask to be able to determine the outcome are:

- How do you know the test case passed?
- What are signs for failing?

It may be evident if the system causes a crash, but in fault tolerant systems even that might never happen.

11.3.10 Clean-up after Test Case Execution

Equally important to create a useful test case is that it should be possible to repeat the test case, over and over. Clean-up after test case executions include all those actions that are needed to be able to remove the effects of execution to be able to execute again.

Observations: Less than 5% of the students attempted to clean-up.

Discussion: This category is easily forgotten, but an obvious category when doing automation. Clean up might contain many actions, and it could be particularly difficult to clean-up in some systems that always store data, and do not allow removal. Software is used for a long time and so are its associated test cases. A test case should be repeatable purely based on economic motive; to allow reuse of the test case and the thought that went into the analysis of the system. In Industry, it is not uncommon that a test case is re-executed up to 100 times within a project, and that it is used for many years. One cannot assume that the person creating the test case will necessarily execute it in the future. Repeatability is to be able to recreate any problem found and requires precise information. It must be possible to check if a specific problem has been corrected afterwards, and the fault is removed. Therefore test cases should not describe a group of data to be used, but should always contain a specific value. And the specific value must be removed after use to repeat the test case. A final aspect of repeatability is to make test cases fast and efficient to execute, typically by automating the execution. In the context of the software life-cycle, probably the cost and complexity of the testware has the same impact as the cost and complexity of the software itself.

11.4 Comparing with Industrial Test Cases

After constructing the categories containing systematic mistakes, we looked at each category and selected a series of industrial test cases and verification specifications, and investigated if any similar mistakes could be found. Lacking full statistical data, we wanted to assess whether if this approach could be taken into industry and used as a means to improve the test case creation. We analyzed a series of test specifications and test cases, and also interviewed and discussed these improvements with a series of managers, testers and developers, to get a more thorough understanding of the results. Our conclusion is that understanding and explaining the mistakes could lead to both improved templates and new ways of working thus improving industrial test cases.

We decided to grade the list, based on our results, into a qualitative scale where the grading for the mistakes frequency is in a three scale range: O (Often), H (it Happens), S (Seldom), In addition, we added one more category (11) and one sub-category (6a), and adapted the category names.

The observed grading for our categories in industrial cases is as follows:

1. Understanding instructions /level of details (**H**)
2. Understanding the purpose of the system and current level and context of testing (**S**)
3. Understanding test design techniques (TDTs) and use them (**O**)
4. Assumptions, e.g. regarding correctness and completeness of specifications (**H**)
5. Only using the most obvious test case or input (**O**)
6. Starting position for the test case (**H**)
    a. Order of execution (**O**)
7. Lacking specification of valid and invalid inputs (**O**)
8. Unambiguous step-by-step description of test case, test execution, and test outcome evaluation (**H**)
9. Not clearly defining the test case evaluation (**S**)
10. Describe clean-up after test case, repeatability **(O)**
11. Separation of instruction and data (**O**)

11.4.1 Understanding Instructions, System and Test Design

This section describes the first three categories of mistakes in our above list. Our first mistake category, the ability to understand instructions and having appropriate level of detail seems to be a pre-requisite for a tester's job. We could see that people accustomed to automated test case creations, as well as developers, were much more skilled in defining details. Still, this category with lack of appropriate detail exists occasionally in industry. Test cases and particularly verification specifications lacked sufficient level of detail allowing for different interpretations depending on the experience of the tester. This happens occasionally in written text, but since most test cases are automated, they possess enough detail. Interviews with the testers revealed that the level of detail is added when automation happens, which makes the automation a rather costly step to take from the abbreviated manual test cases or verification specifications. We could also see a great variety depending on coding skills.

Our second category, lacking appropriate understanding of the system and the test target goal or context is rather rare in industry, and also self-regulating. Experienced testers talked about beginners not making that

connection, and that it was mostly revealed on the type of failure reports written – or shown when reviewing the test documentation.

Our third category, the know-how and utilization of TDTs seems surprisingly low in industry, which could be related to the time pressure, where taking the most obvious test case and inputs dominates. Many testers are aware of this and would like to explore negative testing more, but this is seldom given priority. Most test cases are written in the fashion that they take the first and best positive input to validate (one aspect) of a requirement, and demonstrating that it works. Seldom are techniques utilized to make sure all input categories are explored, such as negative testing using invalid data. By improving the test case templates and verification specification, it would be easier to utilize this result when automating the test cases.

11.4.2 Assumptions in Industry

The assumptions category e.g. regarding correctness and completeness of specifications seems to be earned as a part of one's status and know-how when working in industry. Many testers are rigid, as required by some systems, in the sense that they are following rules strictly, and if the specification (requirement) says so – it must be right. But, if you have confidence and know-how of the system, maybe your first thought as a tester is – maybe this (requirement) is wrong (unclear)? We have also noticed some cultural differences, where some cultures and personalities seem better in confronting poor specifications and, as a result, they create test cases targeting important areas. There is clearly a need for both types of testers.

11.4.3 Obvious Selection of Input values and Test Case

We were rather disappointed that it seems like the obvious input and test case is very common in industry. This result is based on multiple factors, where time pressure to write many test cases and confirm requirements seem to be the dominating one. Other factors might be lack of knowledge on TDTs, not specifying input ranges etc. in specifications.

11.4.4 Categories for Test Case Implementation: Starting position, Descriptions and Evaluations

Definition of a starting position are sometimes missing in the manual or written test specification, where it is assumed users know and understand the context of the execution, and this step is always added to get automatic suites – which are the commonplace type of execution of test. Again, waiting to specify it leaves the problem to the implementation of the test case into executable code. A comment from the testers interviewed is that the test specifications are sometimes written so early, that the specific path to get there might not be crystal clear, and is deliberately left out. What was more interesting is that almost all automated suites were built in a specific order of execution, with rather long series of execution paths. Test specifications in industry could be really large and dividing them in different test cases is common, but they are kept together as a suite. This has limitations, since the technique of permutation, restarting suites at different positions when things go wrong and other benefits of several independent starting positions, are lost.

Therefore we are introducing a subcategory, *Order of execution,* particularly aimed to make automation suites less intercase dependent, with the intention to make smaller, self-contained test cases that could be used in different order in different suits, and re-used with a wide variety of input data.

Specifying the input would greatly improve the utilization of test cases written, and this mistake seems to be commonplace, and we see a lack of know-how translating this into effective test cases. It seems that valid input is more often used than invalid ones since the specifications are being written in this way.

Mistakes in the category of test evaluation are rare. Unfortunately the use of exploratory test seems to influence the perception that clear evaluation is not needed. An unfortunate downside seems to be that these test cases are not repeatable, and thus the know-how and time of creating them are lost. Random execution is a good complement to teach testers the feel and to better learn the system, but expectancy of stumbling across serious faults in our domain is very low. With regards to repeatability of test cases, it seems reasonable to assume that this is paramount for industry. But when interviewing testers, and by looking at some test suites, it becomes clear that it occasionally happens that, a lot of detail and information is missing

from the test cases, making it difficult to repeat without thorough and specific education. The opposite could also be true, that an automated test suite could be encoded with little and no documentation or heading/book-keeping information, making it extremely difficult to update the suite, and thus make it obsolete in a very short time. We have deemed that the specifications of input and its ranges are a common mistake and missing, but to our joy we could also find the opposite. It turns out that the tool QuickCheck [180] has been used, which requires a clarification of the valid and the invalid by defining the borders (minimal – maximal). This is a good step forward for improving the test coverage.

11.4.5 A New Category: Separation of Instruction and Data

The category is motivated by and related to the handling of larger amounts (several thousands) of executed test cases – often directly translated from manual test cases, creating a rather unstructured set of test executions with many overlaps and hard-coded data. It seems that not all test organizations have utilized the possibility to re-structure the test cases. A clear separation of the action/steps of execution and the variety of parameters/ variables/input values would be a great improvement. This would enable a better future-proof and document-minimalistic approach to handle large input domains, or when there is a combinatory explosion of the input domain (having a series of dependent input variables). Instead of making a unique test case for every input, fewer test cases chewing through a series of input-output relations seems to be the most efficient way to automate. This results in a separation of instruction (the step-by-step actions) and data (input). Also techniques like random selection of data can then be used to vary the regression suites. The savings of this approach comes in many forms, by making the test code leaner, handling a variety of input variables and utilizing the test case creation better.

11.5 Systematic Mistake Elimination Method

By our identification of categories of mistakes we realized that even though most categories are very general, the categorization is dependent on the considered application domain and systems, as well as on the experts performing the categorization. As noted when comparing our results with industrial test cases, there may very well be additional categories and/or subcategories that are relevant. Due to this open-endedness of the problem we suggest a meta-approach, which allows the basic method to be extended with categories. Our improvement process works according to Figure 11.3.

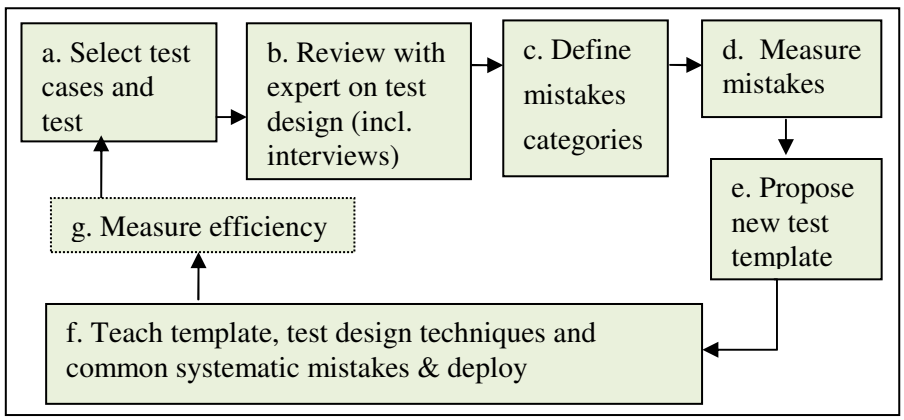


Figure 11.3. Systematic Mistake Elimination Method

This process can be started at two steps, either by assuming the categories in this study as an initial start – or by using the proposed test case template which would be at phase e in Figure 11.3. Otherwise, the first and initial step (assuming that test cases exist) is to select a sample of test cases and test specification or similar documentation where the test design phase is manifested. In step b an expert reviews the different test cases based on how well they have performed different TDTs (which also includes how well the test cases are at targeting faults and contributing to coverage of the system). This will result in the identification of a series of mistakes made.

In c, different systematic mistakes are defined and categories are created, which can then be measured for frequency in the existing test cases. Improvements will then be identified such as a new test template, or usage

of new test case design tools, or improving the required test specifications template. Finally this new know-how must be taught (as shown in phase f), followed by deployment at the organization assessed. The result or improvements should be measurable in g, in a series of measurements, e.g. improved fault frequency, and must be periodically reviewed by the organization.

11.6 Conclusion

This study demonstrates the importance of dissecting the process of test design and understanding details of mistakes made when constructing test cases. It seems possible that testers in industry make systematic mistakes frequently during test case construction, leading to reduced efficiency and effectiveness of the testing efforts, and analyzing these mistakes will form an important basis for improvements in the testing practice.

We have suggested a series of mistake categories, based on analysis of mistakes novice testers made, and we have used this as a starting point for assessing industrial test design. We think this work implies improvements that can be done for requirement formulation (specifically adding input analysis based on the equivalence partitioning technique). Another improvement is focusing on how to better write the test cases in an automated fashion from the beginning (code the step-by-step description, as well as the result evaluation and clean-up aspects), to diminish the effort of translating the often too simplified test case into usable test scripts, where post-processing is defined from the beginning. Creating unambiguous test cases at an early stage also makes it possible to utilize a wider range of the work-force in the execution, which would otherwise become person dependent due to interpretation difficulties.

Making a larger study of a series of industries, using a more randomized sample of test cases, is an obvious next step to get better validation of our method. From an industrial perspective, it would be even more interesting to base on findings from applying our method, device measures (e.g., targeted education and individual feedback to testers) for improving the testing practice, as well as evaluating the effects of these improvements.

Chapter 12. Test Automation

12.1 Summary

Test Automation is one of the main areas for improving efficiency in all aspects of test. It is common to focus on test execution, which has many benefits, since any test suite are often re-executed many times throughout the software life-cycle. Many aspects of automation should be considered in the test design phase, where some techniques work particularly well to generate test cases. Often the problem will be handling the verdict or automating the "test oracle" that can determine the verdict, when generating the test cases. This thesis will not to be handling any of these aspects, but will focus on another aspect of test design – making the test cases traceable to the product and consequently improving the debugging capabilities as well as automating regression test handling.

We present an approach and tool enabling autonomous behavior in an automated test management tool to gain efficiency in concurrent software development and test. By capturing the required quality criteria in the test specifications and automating the test execution, test management can potentially be performed to a great extent without manual intervention.

This work contributes towards a more autonomous behavior within a distributed remote test strategy based on metrics for decision making in automated testing. These metrics optimize management of fault corrections and retest, giving consideration to the impact of the identified weaknesses, such as fault-prone areas in software.

12.1.1 Context of this Case Study

Handling a large amount of automated and manual test cases at different levels of a complex product requires a lot of work from test managers to

keep track of test in different stages of the process. What test cases have been executed, should be executed, have found problems to be solved, and should re-test and who should do what and when, are the real documentation needed in any test management system(TMS). Since the goal of testing is to provide a quality measurement of the test execution, one must aggregate the results of test case verdicts, thus combine the number of e.g. passed, failed and blocked test cases, keep everyone informed about the progress and quality at any instant in time. Our contribution will aid to automate this normal test management process, with the important addition of contributing to automatic fault location, as well as easing management of the corrections and regression test. Our intention was to show that the test design is not restricted to create executable test cases but includes several other aspects worth considering.

Our study is an attempt to describe a proprietary test management system that encompasses some unusual features. It is based on the notion how test design can bind executable test cases to the original source code through the test specifications. This enables analysis towards automatic fault location, and also automates the entire build, rescheduling and regression test. The system features lead to an instant progress reporting and thus contributes to making the test management redundant in the test execution phase.

12.1.2 Design - Research Method

The study is based on identifying test management system functions, and comparing these functions with the TMS tool, providing insight to new aspects of the context and set up of a test management system which are important factors that can provide features that support areas beyond the goal of only the test management itself.

12.1.3 Validity Threats

The study is performed in an industrial setting. It describes a system created and in use for over 6 years without researcher influence in the construction and utilization of the system. The functions and statements about the system are descriptive. This is a post assessment. There are

strong threats to validity, due to the system singularity and researcher bias in reporting about this system.

12.1.4 Contribution

Our results indicate that in addition to efficiency by automating the test execution, automating support for test management beyond the normal test management systems is important. Our suggested automation makes the test execution phase seamless.

We provide the contents of Test Management Systems and expand the automation scope, thereby enabling savings in managing the test execution phase, particularly debugging (fault location), regression testing and test management. The main findings are that

- Setting up test cases connected to software items enables support for analysis of auto-locating faults
- Correct automatic support can save money and time, and could make test managers redundant in the test execution phase

Our aim in this study is to clarify some distinguishing features of our system, and also to present a more elaborate view of future test management and test, which we envisage to be more autonomous in nature, compared to contemporary test management.

12.2 Introduction to Test Automation

12.2.1 Test Automation often means Test Execution Automation

All aspects of the software development – including the testing process – are targets for automation. Today, it is not uncommon to have systems with more than 1 000 requirements, 100 000 test cases, and 1 000 software components, sub-systems and other software entities. These systems are developed in a distributed manner at multiple sites worldwide, include third party software, are frequently updated, and need to be tested at multiple integration levels. For a single site this

could result in more than 50 000 test cases to execute for each iteration/release.

Even in a perfect test execution situation, every test cannot be executed on a daily basis, due to the complexity of the systems under test and execution time of the test cases. The complexity in terms of large sizes and internal dependencies of industrial systems are impacting all aspects of software development and test, something which research is rarely targeting.

To handle the complexity of the test effort an efficient and scalable test management and test execution is required. Current test management requires a lot of manual handling due to the diversity and lack of integration between tools for handling requirements, configuration management (system builds), test specification, test execution, test environments, failure handling, quality and progress reporting, etc. Total integration is not a viable solution, due cost and complexity, as well as its inability to incorporate legacy and emerging solutions. Test Design – creating the test cases for an automatic execution, or for that matter generating these test cases automatically. Evaluation of test case result is an equally challenging task that depends heavily on how the test cases are designed. Test Automation is an area large enough to encompass several PhD thesis' in their own right, our main proposal is showing a factual improvement in the test management/test process execution area.

Test automation is not an easy task. The test automation area needs to mature. Test automation can save a lot of money, and our preliminary studies shows [67] that even within the project time-frame we can shorten the development time, where the break even usually is around the third regression. The degree where it is possible to create an effective and efficient automation is probably dependent on the software under test, but tools and increased know-how make automation of test more commonplace. Today it is not unusual to execute over 50000 test cases in suites. Since daily builds requires a fast turn-around time, there is a need to diminish the test suites which often lead to lessened quality.

12.2.2 Automation of Test Design

When we think Test Design, we have so far been focused in creating test cases, which is the main core of test. Taking test design into a larger perspective and viewing the entire test process. Test design does include all aspects of the handling of a test case, from a system understanding perspective to achieving the actual verdict, result of the test case execution. How this is performed will determine the outcome of the total test efficiency, but the selection and performance does implicitly impact the effectiveness as well.

The automation of a test case consists of several steps:

a. Test case selection criteria (search, generate test cases, create test cases). Could be manual, semi-automated or automated.
b. Test case minimization if automatically generated (based on criteria, often coverage) and completeness (this step might be optional, but could also be thought of as "priority" if more test cases are created than possible to execute).
c. Test environment set-up (manual, semi-automated, automated)
d. Test execution (manual, semi-automated, automated)
e. Test case evaluation (test oracle, manual, semi-automated, automated) often the real hurdle with automation.

The challenges for these steps are:

a. That selection may not work on larger systems. Automatic generation of test cases can be easy, but is often time consuming. Different tactics of limiting the search is often applied. Creating test cases often involves a series of human decisions to minimize and guide the test execution into meaningful test cases.
b. Reduction of the test set is usually no hurdle in itself, since test cases are just discarded if they do not contribute towards the fulfillment of a criterion (e.g. coverage). However, for a complex system, it is difficult to efficiently instrument and measure data-flow coverage and control-flow coverage, or is determined less important for some reason. For small systems, completeness is achievable, but for large systems, there must be criteria to cut, prune or determine when the suite is sufficient. In industrial system, the time available often set very strict delimiters of how much is possible to execute. Unfortunately, more often test is "pruned" when time runs out.

c. Test environment set up is often neglected in the software test processes, but for many industrial systems, this is the most time-consuming and costly part of the test process. It means that there are also a potential to make this phase substantially improved by making automatic set-up of the software and hardware environment, for where the test is to be executed.
d. Execution, from a research stand point, poses no trouble, but in reality, the way *how* this execution is automated could make a huge difference for how useful the automation will be in a real system. Areas that need better support include: Manual intervention (which is often needed for determining test results), logging, tracing, and instrumentation (which are often used as a part of the test case execution), and adaptations and workarounds (to make the test execution possible). Also know-how about administrative issues, such as version handling, traceability, regression testing, and managing, driving and handling the actual tests (and results) e.g. test management will also be a part of the test execution tasks.
e. The evaluation can be problematic for some type of systems, since in a large complex system that often use fault tolerant and fail safe, self managing architectures, a fault is easily hidden and might not propagate immediately and become visible. In this area, detailed research on real systems is needed.

Having a cost-and-time efficient view on all of these challenges, poses a great variety of solutions, whereof we are limiting our view on some aspects of the test design.

Test Automation – including all aspects of the test process are not often considered when wanting to make efficient test. In fact, it is more often that only the test execution part is in focus, but much too often leaving out the actual verdict, which could include a lot of set up and analysis of logs in some systems. It is important as well as aggregating the results as a measurement of the system under test.

In this thesis our goal was to explore efficiency in test design – and all aspects of test that is automatic results in more efficient test. One have to assess every step from system analysis to a maintenance situation when reviewing automation, where our main view is that one must not only look at testing as an individual area, thus it is connected in different aspects to all other work conducted in the software handling, creation, correction and maintenance. Thus – test has a context. In our

study below, we are targeting just such a concept of how the moment of the test design phase – where we create the test specifications, and then apply our TDTs to create test cases – can define a test management system with benefits beyond the normal scope.

12.3 Test Management Automation Study

Development of large and complex software intensive systems with continuous builds typically generates large volumes of information with complex patterns and relations. Systematic and automated approaches are needed for efficient handling of such large quantities of data in a comprehensible way.

In this study we present an approach and tool enabling autonomous behavior in an automated test management tool to gain efficiency in concurrent software development and test. By capturing the required quality criteria in the test specifications and automating the test execution, test management can potentially be performed to a great extent without manual intervention.

This work contributes towards a more autonomous behavior within a distributed remote test strategy based on metrics for decision making in automated testing. These metrics optimize management of fault corrections and retest, giving consideration to the impact of the identified weaknesses, such as fault-prone areas in software.

12.3.1 Introduction to the Case Study

This study presents an autonomous test management framework with the following key features:

- A loosely-coupled integration of diverse tools, based on automated extraction and synthesis of a set of measurements.
- Using test specifications as release criteria, the system becomes ready for release when all tests have been successfully performed.
- Automated traceability of failures to software components, achieved by enforcing test specifications to represent system entities.

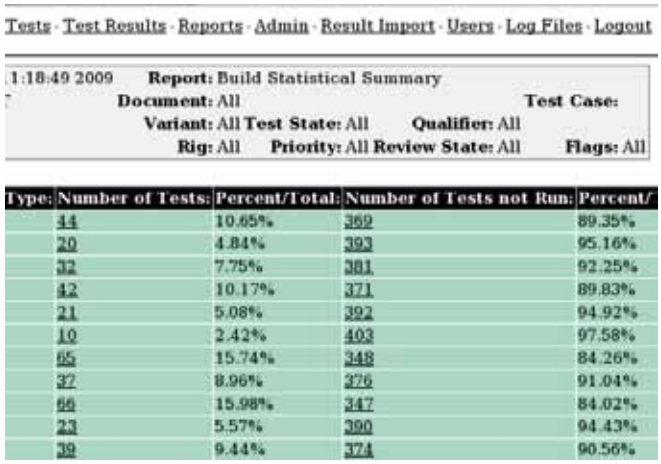
This provides cost efficiency through reduction of manual intervention and improved quality assessment in the context of complex industrial systems, since historic quality data can be taken into account. Using this framework, the role of test managers during test execution can essentially be eliminated. This has contributing to substantial savings for Ericsson over the past seven years in use.

Test management systems have been around for many years, with several commercial off-the-shelf (COTS) tools available. One of the motivations of this work is the problems associated with these, as e.g. outlined in [15]. At least 16 open source systems are available, and within Ericsson more than 20 different test management systems have been developed, considering different needs of automation. One of these systems aimed at tight integration of the involved tools for requirement, configuration, review, software and test management, as well as failure handling, all combined in a large relational database.

Our experiences from working with the above system, as well as with a variety of more loosely coupled test management systems, are that loosely coupled systems are less susceptible to performance problems and are much better in handling legacy systems. Other reasons for our lack of success with a tightly coupled solution, is that test automation is typically achieved by integrating a series of tools, and an integrated solution makes transfer to new tools costly. The flexibility of loosely coupled systems also increases their longevity, increasing the chances of being able to use the same test management system throughout the lifetime of the system.

12.3.2 Overview of Test Management System

Test execution and test management need to cater for the dynamics in the software build process. This means that for large complex industrial software systems the measurements collected throughout the test process are essential for the continuous quality assessments. Test management in the execution phase assumes that test cases are created, and that extensive regression testing is required during the entire project, taking historic test results into account.



Tests · Test Results · Reports · Admin · Result Import · Users · Log Files · Logout

1:18:49 2009 **Report:** Build Statistical Summary
Document: All **Test Case:**
Variant: All **Test State:** All **Qualifier:** All
Rig: All **Priority:** All **Review State:** All **Flags:** All

Type:	Number of Tests:	Percent/Total:	Number of Tests not Run:	Percent/
	44	10.65%	369	89.35%
	20	4.84%	393	95.16%
	32	7.75%	381	92.25%
	42	10.17%	371	89.83%
	21	5.08%	392	94.92%
	10	2.42%	403	97.58%
	65	15.74%	348	84.26%
	37	8.96%	376	91.04%
	66	15.98%	347	84.02%
	23	5.57%	390	94.43%
	39	9.44%	374	90.56%

Figure 12.1 Overview of progress (number of builds and % of test cases executed)

The historic data can be as recent as yesterday, or originate from substantially older versions of the system. A positive consequence of automation is that it becomes feasible to mine and collect new measurements, which give new insights to the efficiency of software development and test, as well as highlighting potential flaws in a chosen approach. Figure 12.1 shows a screen shot from a test management tool, presenting the fraction of test cases executed for a sequence of builds. This is a typical picture of a parallel development and test process, showing how configuration management is combined with monitoring of test status.

The test management process typically consists of a test preparation phase (Figure 12.2) and a test execution phase (detailed in Figure 12.3). In Figure 12.2 clouds denote manual work. In Figure 12.3 all tasks contain manual work in a COTS tool, whereas in our tool writing of failure report (by choice) as well as analyze (debug) and correct the fault, are the only tasks that still contain manual work.

The test preparation phase is often the same in different test management systems, although with variations in how data is stored, e.g. the test specification can either be a separate textual document or an entry in the database/test management system.

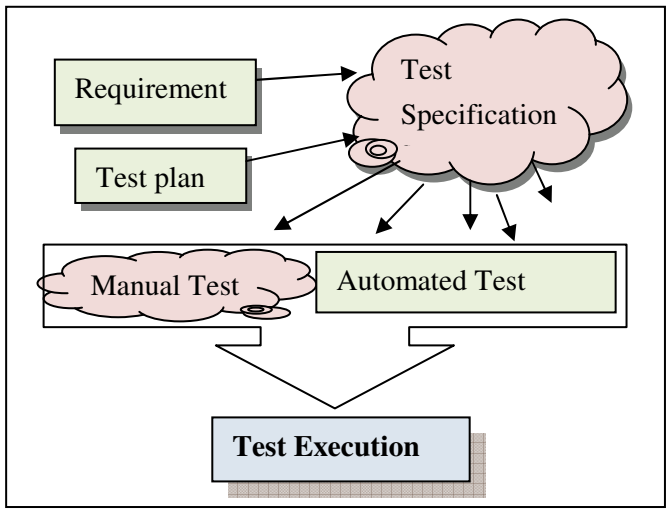


Figure 12.2 The Test Preparation Phase

The test specifications might be traceable to the requirements, and implicitly, the test case may be indirectly traceable (usually through the ID or naming convention) to either a certain test aspect or to a requirement. Some test management systems can only provide this traceability if the requirements are in the same system. An alternative is that the relations between tests and the requirements, and thus traceability, is provided in the requirement management system, requiring the testers to enter information in more than one system. It is clear that in neither of these systems there is a direct relation or possibility to trace test cases to the actual software being tested.

Basic data that are possible to record, edit, view and store in any test management system are the following:

- Dates, creators´ information which includes *version handling and change information*
- *Test cases*, including: Test ID/name/slogan, priority, and type. More elaborate systems allow a step by step description similar to the test instruction/test procedure assuming a manual or key-word driven test. Test Management systems do not contain test execution capabilities, which are left to the test automation system.

- *Scheduling* of tests (planning) which means assigning testers and other resources to one or more test cases, with qualifying information such as time, date and by whom.
- *Status of the test execution* (captured from the automated tool, imported from the automated system or manually recorded) such as passed, failed or attempted test cases. Variants of these status fields exist [60]. For failed test cases, often textual fields that can be used for recording Failure Reports name or ID (e.g. failure ID from a failure tracking system).
- *Test environment/test bed* information, which is either a link to a document, a slogan/ID or a textual field.
- A number of *reports*, which provide possibilities to view the stored data in different ways, presented in graphical form with trend analysis.

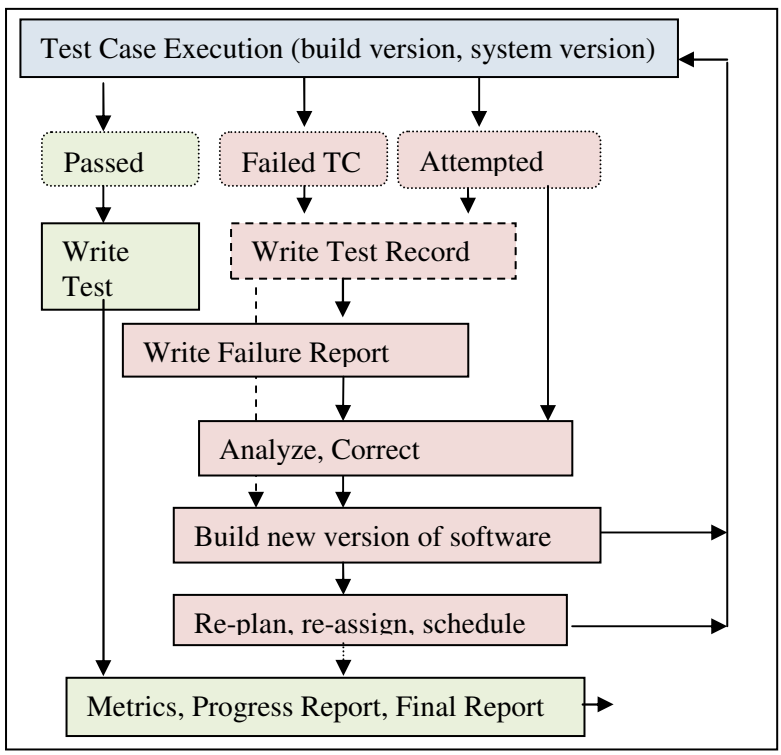


Figure 12.3 Activities within test execution

Depending on the test management system, different philosophies exist on what and how much is stored in the system. In some systems, only test cases and their status are stored, and others store the entire process [177]. This means that either a certain number of documents are to be linked or need to be directly written in the system. Examples of data that should exist can be found in IEEE Std. 829 -1998 [110] i.e. test plan, test specification, test procedure (often implemented as test case), test record, test reports, which all would be textual. Finally the tools have built-in or indirect methods to integrate with other systems. For example, integration with a failure tracking system or with a test automation system is common, but more advanced test management system can also be integrated to support traceability with requirements management systems.

12.3.3 Limitations of Current Test Management Systems

There is a series of problems with the current approach to test management that are listed and explained below:

- Information handling of test cases are manual [102] [115][170] [178][180] [197]
- The test cases are often written assuming manual execution or a tool within the same tool family [102] [170] [178][180] [197]
- Lack of version control or communication with a configuration management tool (outside the “family of tools”). [102] [115] [170] [178][180] [197]
- Collection of information from other systems(failure tracking, configuration management, test automation tools) is often manual [102] [115] [170] [178][180] [197]
- Historic data (traceability to the software quality) between projects is not available [102][106] [115] [170] [178][180] [197]
- Changes in status of failures often requires manual imports [102] [106] [115] [170] [178][180] [197]
- Scheduling, starting and management of regression testing (what test cases to select) is mostly a manual task [102] [106] [115] [170] [178][180] [197]

Most of these systems deal with management and handling of *manual* test cases, their execution and outcome, and all information is entered into the system by hand. If the test case is automated, it is not very clear what and how much of the managing information that is in the test execution automation system or in the management system. Automated test execution systems often have rudimentary handling and management of the test cases, and provide verdicts, often in the form of logs. These systems often fail to collate and present the result without substantial manual work or without manually exporting data to another system. Historic data is not available without extensive data mining and without know-how of the software system structure in the different projects.

Handling of failure tracking in separate tools, e.g. [76], requires testers to constantly monitor when the correction is scheduled into a build. Though more modern failure tracking system can inform when a failure has been debugged and corrected, a lot of manual interventions are still required, both to make the new version with the corrections and to create information about the corrections included. This is tedious manual work. Not only must the re-assignment be done, but the actual planning of what test bed and what build to use etc. are manually selected and the build might be manually initiated. When the goal is a daily build process, this would beyond doubt require full time work to supervise, to make sure what corrections are submitted in the build, and what test can be executed as a consequence. Handling a large amount of automated test cases, the problem becomes unmanageable, and often solutions such as minimizing the number of test cases are adopted to make sure a decent turn-around time is met. Minimization of test suites seriously compromises the overall quality of the test. Current test management tools lack in active and seamless integration with tools for system build, failure tracking, automated test execution, and report construction. Ericsson has attempted to create an “ultimate” tool, in which all involved systems are retained. Unfortunately this resulted in heavy performance problems, and lack of scalability between different sized projects. This system also enforced a rigid (inflexible) development process. A similar though more open approach is adopted in the new Jazz platform by IBM/Rational [106].

12.4 Our Test Management System Solution

Our Test Management System (named TMS) enables a new approach to automated test management. In TMS (shown in Figure 12.4) we introduce the following new elements compared to standard test management as shown in Figure 12.2: (1) we make explicit links between test specifications and elements of the system structure, (2) we use test specifications as release criteria, and (3) we make sure the manual test cases can be handled by the test management system, rather than being contained in separate documents. This enables control of execution through sending email to a tester when a specific manual test is to be executed or re-tested after a correction.

By defining the test specification as release criteria, and by letting test specification represent a part of the software (could both be aspects of the system, or a particular part of the software, e.g. a sub-system, component or similar), it becomes possible to use each test specification as a container for the test cases defining the particular release criteria for that particular part of the software. Each part of the software is uniquely identifiable through its structure, naming and hierarchy.

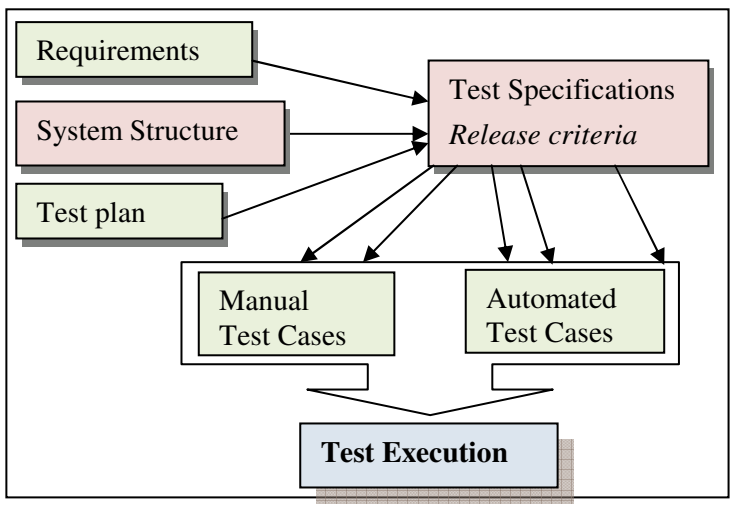


Figure 12.4 The Test Preparation Phase in TMS

The unique ID is denoting each source code, each executable, and furthermore each "level" of integration. The unique ID of the software in the system structure makes it possible to link software with the test specification. We create our test cases as an implementation of a test specification. The test specification is then manually related to the test object list, which is in its turn related to the software structure. Hence, traceability will automatically be obtained for the test case. The binding of the system structure to the test specification is indicated by the pink areas in Figure 12.4.

By tracing test result back not only to specific requirements, but also to the software system, it becomes possible to retrieve historic data on software quality. This gives a better overview of quality over time, compared to any other available test management system.

12.5 Is TMS a Fully Automated Solution?

The test execution can be fully automated, though we claim that there is currently no business case for fully automated generation of failure reports. The reason being that if a test case does not pass, a manual review must be made at some point in the process anyhow, and if this is done early in the process, our experience is that a substantially smaller number of failure reports are written.

Our estimations indicate that each duplicate of a failure report consumes on average one hour, and in one project this could easily amount to several man years in extra costs if allowed. This manual handling of failure reports is labor intensive, and research contributions can be seen in e.g. automatic debugging [2][220], but have not yet reached industrial maturity.

For example many test cases can fail for the same reason, causing duplications of reports, and it is possible that a failure is already reported in another failure tracking system or for another project on the same software. Instead of analyzing a series of "duplicate" failures that might have occurred due to the latest change, and same source file, we perform a brief analysis of the failed test cases by the testers in relation to software involved, minimizing the number of failure reports. If the same failure occurs as the result of multiple test cases, then only a single

failure report should be written. The relations are visible in Figure 12.5[4], showing that failures are associated to sets of test cases.

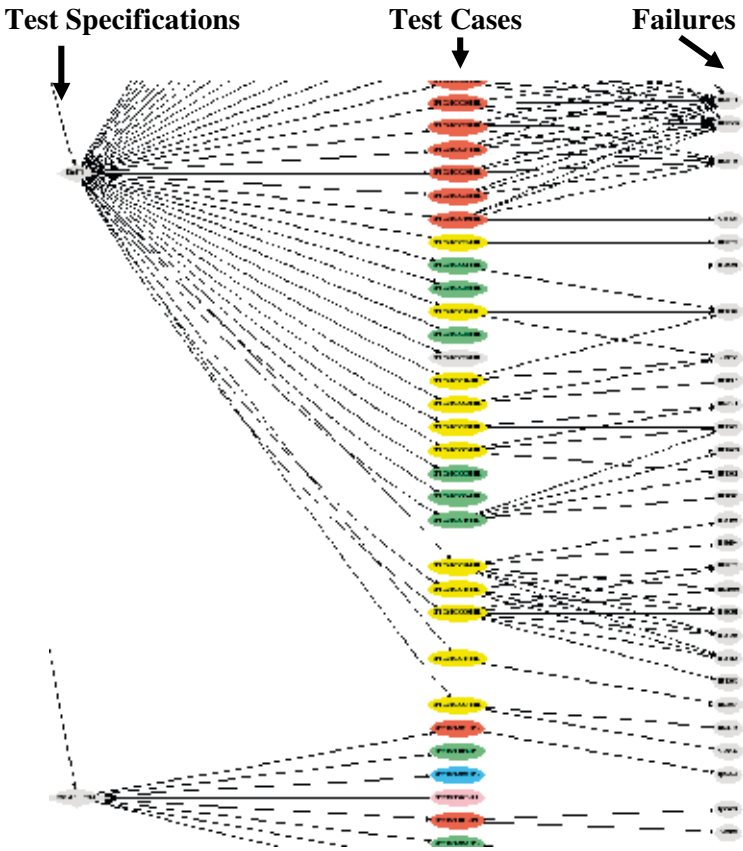


Figure 12.5 Hierarchical/traceability view: Relationship with failures, test case and system

This is saving considerable time for developers correcting the corresponding code fault. Consequently, when failure reports are manually connected to test cases, the system makes an automated

[4] Note that this is just an example in which actual content (viewable graph) in TMS, in which individual test specification, test cases, and failures in the figure are not supposed to be readable.

association to trace a failure to a specific software unit identified (since the test specification is a direct trace to the software). This feature in the system supports faster identification of who is to debug and correct the problem. This is an essential time saver in globally distributed software development.

12.6 Test Case Scheduling

The total number of test cases developed for a particular product should be represented in the test management system. The database containing this information is very large, and similar to most test management systems. The active selection of what test cases are going to be used and updated for a particular project (software enhancement) is still a test manager task. Once it is decided that these test cases form the right scope for the project, and the test cases are assigned to a tester, the test management system takes over and scheduling is automated, meaning that if a test case has failed it will be automatically re-scheduled when the software is corrected in a new build.

Any management information needed will be retrieved by pre-defined queries to both the proprietary and commercial systems used. For example, different types of failure tracking systems are checked by the time of a request. Some data in the different databases are actively refreshed, and some need page-refreshment to be accurate. E.g., when a failure report has been “fixed” or cleared by the developer, TMS will first identify the status-change and then trigger the automated rescheduling of re-execution of all the test cases involved in that failure report. As a user you can set this regression testing at any frequency you prefer.

Our system can concurrently work with several different failure tracking systems. Traceability with respect to the initial requirements is provided by the system (see Figure 12.5). The test case can be tagged for regression test, for use in a particular version or branch, or for other purposes. The test cases are as usual in a test management system containing author, date, updates, priority and other necessary information.

The difference here is that the test execution will be performed by a diverse set of test tools, and that the test management system is able to handle the results in an integrated manner. The aim is to group the test cases according to how the tests relate to the structure/anatomy of the software system into software component or series of integration components. The hierarchy of the software system is clearly reflected in the different levels, as illustrated in Figure 12.5. By hierarchically organizing the test cases, it is easy to select which test cases should be used for executing a particular part of the system that has been changed. From an automation standpoint, TMS provides available test cases for a particular software product (part) and makes it easy to define and decide which test cases should be planned for execution and on what test bed/test environment.

12.7 Test Execution

Test Execution means that the test procedure or test cases are automated (implemented) in some script language to be executed with given data. Currently the test suites are driven from the test execution system, but with little effort it is possible to glue existing test harness systems with the test management.

Once the system is set up at the beginning of the project, the execution is fully automated. Still manual test cases might be assigned and executed in parallel with an automated suite. Invoking regression test (after a fix, or when initiating the system) is triggered by the TMS sending an email to a set of pre-assigned testers. Not all test cases are possible to automate (at a reasonable cost), since some tests contain elements of manual nature, e.g. to test the behavior when a cable is broken (simulated by a pulling it out manually).

This means that our automation can combine and integrate the test execution, the build system and the failure tracking system, and yet allow these systems to develop and mature at their own pace, to achieve the best execution. A test management system that combines supervision will provide a better overview of status, e.g. checking in which state a particular failure is, and have a convenient overview of what is tested in each version and in what build, in addition to provide system specific quality and historic data.

12.8 Instance Progress Reporting

The planning of test cases in TMS provides few new features compared to existing commercial software tools, since creating test cases, naming and describing them in a stepwise manner, and provide a verdict is not new. However, by deliberately making test specifications a representation of the system and requirements, the set of test cases will represent the system quality criteria and thus function as release criteria. This eliminates the need of monitoring during the execution phase, which substantially improves efficiency of this phase. The test management system will in addition to planning, also support the manager when deciding which test cases should be run at a specific instant, and then it should collect the result in a way that makes the evaluation of the system clear. These common tasks are normally handled manually, as is the re-assignment to the next test execution of test cases that did not pass. Often test managers need to spend most of their time in following up problems, in particular related to the test cases that did not pass. This means ensuring that someone takes care of the problem, analyzes it, debugs and corrects the fault(s), and then makes sure that the proper regression testing is done. Many systems will provide a clear pass or fail picture of the system at any instant, but there are seldom more information that improves on the quality assessment. A more modern test management system allows defining the status types at set up time. Combining historic and concurrent information of software components is an important feature of this system that contributes to a much more elaborate analysis of quality and regression tests. Not only should the particular test case which found the failure be re-tested, but also any other dependent test cases should be re-scheduled. How the dependencies of test cases are related to the software components is described and defined in the test specifications and test procedures. This feature is provided by the system and will in both a push and pull fashion get the test result from whatever test automation tool in use. The feature is one of the main contributors to the perceived savings and success of TMS. The hierarchy of the system and its test cases is clearly reflected – and visible – in the test management system.

The most common data used to describe progress and actual results for most Test Management systems are combining the accumulated number

of test cases planned, attempted, passed or failed, as well as faults found and fixed. In addition Ericsson uses the following status labels:

- *Blocked Not Run*, which means the test case cannot be executed, since something (failure of another test case, code, equipment etc) is missing
- *Blocked Run*, which means that the test case was attempted and executed partially, but could not complete due to a blocking problem (special case of fail)
- *Concessed*, which means that the test case was granted a concession to be deferred to a later release due to irresolvable reason.
- *Not Possible*, a status describing test case description/implementation that was scoped in but not implemented satisfactorily to test execution
- *Correction (fix) Available*, which is a measure of corrections (that should fix a failed test case), as discussed with respect to fault-failure relations in [66].

The combined test case results for a certain project is also available at any time, which supports baseline, and release support. This means that at any given moment it is possible to enforce sign off-procedures, which is to be used for auditing purposes. This status is communicated by the use of icons and color in the system.

The trend analysis can be used to monitor progress at any time of the test execution, and gives information on how much work is needed for completion of the test tasks in the project. This is considered the most important task of a test management system. We have found that this data can give an accurate picture of the progress of the software development, but it can also be deceiving since the reporting is based on a non-disturbed environment, and contains accumulated data. In fact, for every new build or release there should be an individual curve, since each build is a "new" system, and if not a thorough dependency analysis (on how the software impacts the change) is done, one cannot make sure that because a test case passed the first time, the same test case is unaffected in the next new build version. The probability that the test case will pass the next time is probably high, but should not be taken for granted. Therefore it is important to have other metrics that give a better picture of the quality. Currently the Configuration Management system holds the data of what lines have been changed, but the information is

insufficient, since every code change triggers a series of execution impacts, and can thus make test cases that have passed fail, even if they were not directly involved in the change. This is due to dynamic binding and dependencies in the system.

There is seldom time to execute all tests in a regression suite. Based on available reports we have identified that some projects could not execute more than 40% of their test cases in a suite. A goal must be that at least once in each release, all test cases should pass, preferably in the final test suite. By comparing the accumulated view provided in the trend analysis, the percent of tests provides a new visibility, which provides an overview of the system regression testing can be reached. It should be clear that what is considered "100%" completed in Figure 12.1, is referring to the selected number of test cases, not the system. The numbers of selected test cases are still only a small sample of all possible test cases that could be created for the software in question. The number of builds in a system is an indication of the number of changes of the system.

12.9 Failure Tracking and Test Case Relations

TMS allows different failure tracking systems to be used for the same project. This reflects a common work-habit that developers in the agile work-teams often record their failures differently than testers. The customers also have different ways to report anomalies. The different failures should be consolidated by relating them to one or many test cases. If the test case is hierarchically organized in containers reflecting the system components, this could give a strong indication on what part of the system needs more attention. One example of this is that if a particular fault is contributing to failing several test cases, it should be given priority.

This support is a necessity when there are thousands of failures to correct in a large complex system, where this still equates to a failure density of telecom grade (aka 10^{-6}) in relation to the code base. In this context it is not humanly possible to determine the priority in terms of failures relating to the development. Our system contributes to an

improved sensitivity analysis of the failures and subsequent software improvement, due to better data.

12.10 The Test Management System Used In this Study

Test Management System (TMS) was developed by using open source products, including MySQL v. 5 [14], Apache web server v.2 [15], and Perl scripts. The first prototype was developed in 2002 during one year, and has been used and constantly improved since then. TMS is an example of a tool that has been created out of need, in a situation where no commercial tool has been possible to use or available to handle the diverse environment. By integration with the largest test harness execution system at Ericsson, it will support future generations in strong competition with a number of commercial tools.

Is TMS fully automated? Yes, in one sense, since all the tasks performed by the test manager in the execution phase can be removed. The entire test execution phase includes manual tester tasks, e.g. writing failure report, to minimize the administrative overhead in analyzing (see discussion below). The idea was that once the test cases were entered, the system would be available to all to give an instant overview of all aspects of the progress until completion through its web-interface, guarded in availability by user security settings. This goal is not farfetched, and few enhancements are needed to get there. We are still hesitant to automate the failure report writing, even if it was possible, since all available logs and data exist. We can easily assign the failed test cases to the correct design team, but believe that it is better to let testers write the reports. But even if this part of the process is manual, the rest of the process is automated, since when the debugging (currently also manual – but not a part of any test management system) and correction is done, the developer responsible will assign the correction to a build and change the status in the report.

Currently the test management system will monitor and make sure the correct test or tests for the specific fault is rescheduled and retested. This is visible in Figure 12.5, where it is easy to identify test cases that failed, and the current status of the correction. Figure 12.5 also shows that one test case can have two or more failures that must be corrected

to pass the test case. The colors represent the status, e.g. if a passed test case have a failure associated with it but is now corrected and re-tested.

The TMS system contains a series of reports, not seen in normal test management systems, including:

- Test Statistical overview (Pie chart)
- Hierarchical/traceability overview (Figure 12.5) – where you can visualize the relationship between the system/documents test cases and failures.
- Test State overview – which gives the test progress by Test Specification, thus software area or characteristic of the system.
- Failure (Fault) Statistical summary– which gives failures based on priority – where you can get a multitude of system versions, and also for all history of the product, where different states can be identified from the failure tracking system.
- Time Statistical Summary – which gives time for executing the test case. This is automatically recorded for the automated tests, and gives a feedback for planning purposes. This also gives an input on how to combine automated daily builds in different suites. It also shows, the total time used for running a certain test, and an average execution time for each test.
- Change Activity Report – presenting what a particular team has done recently, and what change has occurred in the project.

In our quest to automate the entire management, we want to find measurements that truly would give important insight in the actual process. Not only how progress is proceeding, but giving us a feedback on the quality evaluation – and also on where extra attention needs to be focused on. Important data that could easily be retrieved from the TMS system are:

- What test cases are associated with a certain part of the system, and traceability to either requirement or software parts
- What faults/failures that are known for a certain product – and the current status of all these
- How many faults a certain product or part of product have had over time and where
- How long a test execution takes (both for manual and execution), which make better planning of regression suites and manual testing possible

- Estimations on total time for test execution in the system
- What test cases located failures
- Time to fix a fault and time for retest e.g. turnaround time

These measures contribute to improve planning of both manual and automated tests, planning of order to correct failures – which is a very important area in large complex system, since more testing means more failures found. An advanced test suite finds more failures, since it has higher coverage.

12.11 Discussion

Our claim is that it is possible to automate many of the test management tasks, e.g. supervision of test cases being corrected and reassigning test cases to the different testers. This means that the work of the test manager could diminish almost completely in the test execution phase. The tester, once assigned to the test case, will make sure it executes to completion, analyze if a failure occurs, and report failures, and when corrected, automatically be reassigned to re-testing (if manual), or alerted if a failure in the automatic suite is occurring. The entire test suite will be automatically executed for each release. Automating the test case execution will in itself contribute to efficiency, but our focus on automation of the entire test execution phase could save substantial additional time.

We have built and used a tool in real production, and tried it over seven years to evaluate its sustainability to carter for old and new tool adaptations. In practice, a large number of tools are used to cater for different aspects of the test process in large scale, complex software development within Ericsson.

Our results indicate that not only does efficiency by automating the test execution, but also by automating support for test management beyond the normal test management systems, which makes the test execution phase seamless. Our aim in this study is to clarify some distinguishing features of our system, and also to present a more elaborate view of future test management and test, which we envisage to be more autonomous in nature, compared to contemporary test management.

The strength of our test management system lays in its open interaction with failure handling, build management systems, and several different test automation and execution systems, which enables new types of automated information processing. During the setup of the test plan and test specification, traceability is provided through the test specifications that explicitly relate the requirements to the corresponding software system parts. This is enabled by active polling through queries; the result of the query (the log) gets selected, aggregated and displayed according to the presentation required in our test management system web interface.

This contributes to a new and unique overview of the system being developed. The data can then be combined into reports containing, e.g. execution times, failure traceability and test case dependencies, which adds new information and insight into the product development.

Development of the TMS system as well as its evolution over the past few years has given us a series of insights. We have questioned the practice where current test management systems require many manual tasks. Our insights can be summarized in the following statements:

- Creating test cases based on test specifications that are defined (linked) to the system components enables traceability. This provides the novel feature of being able to assess the quality of individual software components over time in many different versions and configurations, instead of only for a specific project.
- Considering a set of passed test cases as release criteria makes the test manager role redundant in the test execution, since all tasks of are handled by the test management system.
- We can automate the test execution process, including the handling of build information, and failure tracking and to combine these different tools and separate processes into one seamless flow of loosely coupled separate systems.
- Even if it is possible to automate failure reports, we advise against it. If one fault makes many test cases fail, this will result in many duplicated failure reports, which is inefficient.
- It is possible to automatically build, reschedule and automatically re-test corrected code, without any human intervention. This enables daily build and an extremely quick turn-around time.

By our contribution several manual tasks in the test execution phase are not needed in our proposed test management system, such as handling progress, handing out tasks, dealing with failure reports, re-planning of test cases for regression test, and identifying software with poor quality.

The remaining tasks can be part of the basic project management. Progress and trend information is available instantly in the system. The system is ready for release when all test cases are passed or remaining failure cases have been handled. Test execution planning of the regression suites improves when execution times are available for both manual and automated test cases.

Tracking execution times will be instrumental in planning the daily build regression suites, to make sure the test suite can be completed within the time limits. Our system contributes to remove some of the administrative overhead, where one important aspect is enabling a faster turn-around time (the time between failure discovery, until it is corrected and back for retesting).

12.12 Related Work

There are several commercial off-the-shelf (COTS) tools such as [102] [102] [106] [108] [115] [170] [178] [180] [197] available. One of the motivations of this work is the problems associated with these, as e.g. outlined in [15]. There are at least 16 open source systems available with various features and qualities identified [170][7], and within Ericsson more than 20 different test management systems have been developed, considering different needs of automation.

There is a web-based system to support a process similar to ours [83] and a related evaluation of test automation management [80]. None of these solutions cater for all aspects in our solution, especially the handling of a multitude of different systems. The area of test management tools is surprisingly scarce in research studies, as vivid as it is in usage, especially during the latest decade, e.g. [60]. The common view is that test management is either a result of automating the test process [177] or a consequence of test execution automation [23]. Defining the set of test cases as quality criteria up front, i.e. by letting successful completion of all test cases in the set be the release criteria is an old idea [31], and defining test specifications as the bases for release

is not new. Organizing the test specification in a way that reflects requirements is the most common approach, supported by standards [110] that yield traceability [165], Connecting test cases to source file is also part of the regression paradigm in [213]. The notion of grouping the test specifications in direct relation not only to requirements, but also to the different software entities is new. Traceability is always possible to set up manually or automatically. Technically this is not a problem, but creating the active relations is seldom done. Instead most organizations define this to be the prime work of the test manager. The new system Jazz [106] attempts an approach similar to ours, by requiring all other systems not already within the framework to add an *eclipse* plug-in. This solution might be a bit costly for some legacy tools. The use of the *eclipse* framework [59] is an improvement, since it is enabling tool integrations, which provides a partial solution to our problem.

Currently there is little support for test design in any test management system. Only some of the TDTs are supported by tools for test case generation, e.g., evolutionary/genetic programming (GP) [206] systems and state transition/model based testing (MBT) [45] systems. This is today entirely an engineering task, even if formal approaches can solve specific task in such systems. Even a very small software component in our system will be at least around 200 000 lines of code, implying that the use of only MBT as the main source of TDTs is not sufficient for a quality test [66].

12.13 Conclusion

This study demonstrates the importance of continuous data capture of test results in relation to traceability and failure detection, as a result of an automated test process. This is an important step towards a fully automated test management, through the following main features:

- A loosely-coupled integration of diverse tools, based on automatic extraction and synthesis of a set of measurements.
- Using test specifications as release criteria, i.e. the system will be ready for release when all tests have been successfully performed.
- Automatic traceability of failures to software components, achieved by enforcing test specifications to represent system entities.

By the development and use of a test management system (TMS) that can integrate a wide variety of external systems and thus automated the entire execution, correction and re-testing, we have demonstrated the feasibility and value of automation of test management. We believe we have contributed towards an autonomous solution, and thus minimized the work for test managers in the execution phase.

Part III
Synthesis

Chapter 13. Test Design Techniques

13.1 Introduction

This chapter can be considered a summary of the body of knowledge on TDTs. In addition to our studies, some of these results come from 25 years of practical knowledge in the field. We are presenting the TDTs not on a very detailed level, thus assuming some basic knowledge. We have considered the literature on these techniques and are particularly influenced by the work of Vegas [203] and Murnane [157][158][159]. We propose a test design technique taxonomy that we believe is easy to comprehend. By presenting different ways to group the different techniques we provide perspectives intended to facilitate the understanding of differences and similarities. We have also compared our views with a recent book by Ammann and Offutt [4]. Furthermore, we discuss a series of concepts that is typically used in conjunction with these techniques. Finally we present a subsumes hierarchy showing that some technique are fully contained in other techniques. This hierarchy helps us understand the relations between different TDTs.

13.2 Structuring TDTs

First we present some basic views that form the foundation for how we organize the techniques. Our overview of some TDTs is provided in Chapter 2.7.

13.2.1 Observing Specified vs. Implicit and Unspecified Behavior

We first divide the TDTs in two major groups depending on its relation to the specified behavior of the system; what is implicit or not

described behavior in the specification. If all behavior that is specified is well defined – we have a system that is possible to verify by formal proof. In most industrial systems, a specification is a step in the development cycle, with more and less well-defined statements. There are of course many "levels" of refinement in complex systems, where e.g. modeling can be a great compromise between informal textual description and more formal specification. In the V-model, requirement specification is often the first step that then can go on to a successively more detailed design description before being implemented to code. In the W-model [87], we also take the test requirements into consideration.

To simply verify a specification – or create test cases to show that something works, by just executing the software – seems to be the predominant approach in industry for any software system. The goal is to describe, check, and show or demonstrate that the behavior is what is expected according to the specifications given. This test technique can be called "positive or specified" tests. Some even refer to them as "normal tests", assuming that some normal execution flow (purpose) can be described with the system.

The orthogonal group to specified or positive tests, can be called "Negative or unspecified behavioral" tests, where the goal is to find areas that are not explicitly specified (thus implicit). In practice this could be to enter input that would be "not allowed", or execute unspecified combinations of paths in the system. This TDT is often secondary in priority, and as a result will more often cover untested and unused parts of the system. Thus, the likelihood of finding failures could be much higher. Negative tests have become more interesting to exploit, when software is now more open to the public, and also makes it possible to exploit weaknesses in the software for malicious behavior. These two "high-level" techniques have a main problem: they are too abstract to be useful.

One can essentially produce almost any test case and claim that it is showing or demonstrating the software and hence is a positive technique. This causes in itself a confusion of definition and terms. By instead trying to identify areas that have similar traits, maybe not only resulting in the same test case, would be a sufficient criteria. Also that techniques target exactly the same domain, even if the representation would be similar, seems like a starting formation of our taxonomy – finding synonyms and identifying variants of techniques.

It is common [127],[194] and plausible to use the interpretation of "positive" as only denoting the test cases with valid executions, where valid is to be interpreted as executions that are using valid input, or valid order that does not invoke error messages.

Due to the fuzziness of what really is "valid" to a system under different circumstances, this might be dependent on technique and system, which is not adding clarity. Furthermore one can argue that *all tests* are positive tests, if the system is well specified. Thus, if we specify error messages and what should happen, given "any" false input and any odd combination or order, these should also be (in our definition) considered a positive TDT, which many persons might consider odd. We define the positive technique in the manner that all aspects (even a failure situation) that are well specified constitute a positive behavior.

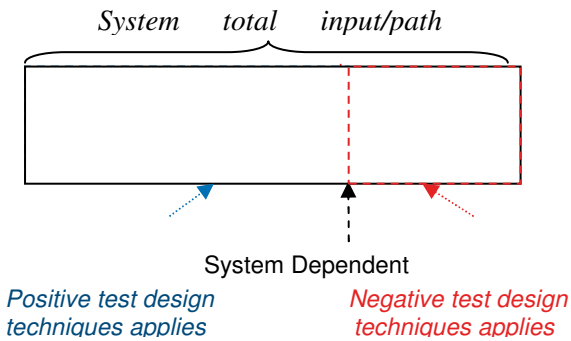


Figure 13.1 Influence of the level of specification of system behavior on applicability of TDTs

Thus, the red area in Figure 13.1 describes the implicit areas of the system and thus is target for what is unspecified in the system. A technique targeting this area is a negative TDT. This has nothing to do with the values being "negative" or if there is an error in the code, which seems to be a normal mistake of beginners in testing. It could be that the system limits are not defined, thus testing outside the scope of the system should be considered a "negative" TDT. .

For safety-critical systems almost no aspect should be unspecified, and systematic approaches, e.g. FMEA [151][149], have been created to target as much as possible of the unspecified behavior and transform it to specified behavior. This is a strong failure preventing

technique – which also shows that well specified systems are less likely to contain faults. A particular problem arises when there are no written specifications or documentation about the system. Defining what is valid, normal and specified falls flat, and one need to make plausible assumptions about the system, which always can be questionable. The only time one can be sure that the system not working properly is when the system crashes, or data or system becomes irrefutable obscure in its responses. This opens up for a more "opinion"-based set of tests, where the tester must have know-how of its intended usage, to make reasonable judgments and assumptions. In this type of testing, the role of the tester is more to define the limitations of the system, than to check that it works according to some other notion.

Figure 13.2 is an attempt to visualize the overlap of techniques in an **existing system**.

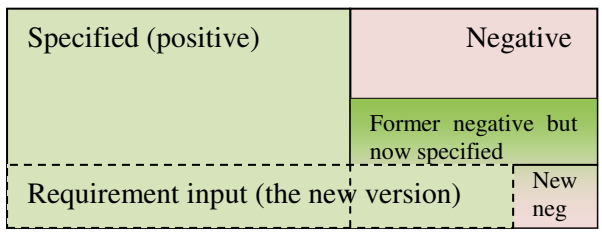


Figure 13.2 Hierarchy of TDTs for an existing software system

In this example the overlap of techniques are visible. All negative test cases (red) that has been defined and specified in earlier versions are now positive (specified). The requirement only targets a fraction of the system that needs some form of change, including adding new functionality that also introduces new unspecified conditions that can be targeted by negative TDTs. This requirement specification does specifically add robustness test cases, and thus target negative test case that now becomes specified (in green).

13.2.2 Observing Input and Output Behavior versus Organization and Structure

Another division or grouping of techniques could be with the focus on data and path. Since data as input implicitly determines the path, this division has its restrictions, and overlaps can easily be found among the different techniques classified in this manner. Therefore it must be clear that the primary focus of the test technique is if a test case is based on data selection or path selection. It is easy to give an example of a technique that deals with both, for example the boundary value analysis technique, which identifies and divides input data at a borderline, but behind the borderline is usually an "if then" branching statement. Selecting input data will yield a division of testing branches (as well as testing if the selection on data was set at the right border). To select data as input with the intention to create a test case for a specific path (in the code) is then addressing the structural aspect.

13.2.3 Comments about Test and Test Levels

We have in our research explored TDTs with regard to levels. Traditionally, three levels exist: code level, integration level and system level. In our research it has become clear that these levels do not form a good differentiator for TDTs, **since we claim that all TDTs can be applied at any level.** The difference lies in the goal of the test, the skill set (know-how) of the person performing the test, and actual "time" limits depending on the system. Our conclusion is that these levels instead provide a **way to organize** the testing during development. For complex systems the number of systems of systems and integration levels are so many that new names are needed. For smaller systems only system level (user level) and the code exists as two distinct levels. Integration test makes little sense if the system is not modularly built, or has clear and distinct conceptual units or sub-systems forming the system.

Additionally, from a testing perspective, there are three clear perspectives that are not the same as levels. One perspective is testing an executable. Every executable has an interface – at all levels and for all testing – through which the system is tested. The other view or

perspective is testing an aspect, behavior or specific characteristics. This often requires repeating different test patterns in different contexts, and analyzing the result. The more you know, the better your analysis is. As a final perspective one can be looking at the code, path, structure and/or implementation as a guide for how and what to test. This assumes know-how of code, and can of course be done by isolating an execution path in the code, or by specifically aiming to execute a code path. These three perspectives are useful at all of the traditional levels of the system. This has been an insight as a result of the study in Chapter 7 and confirmed again in the study in Chapter 9.

Usefulness of test (efficiency and effectiveness) is another matter. It is clear that people with more skills and understanding of software, code, and the target domain, will always produce better test cases that targets essential aspects – if not going too "home-blind", which might be a consequence of too much familiarity.

The main argument for why "system" is not a well-defined concept is that anything can be a system, since it is delimited to what we define as the scope of that system. When a system exists, it is impossible to differentiate the parts and components. A component can be as small or large as we define them to be. Usually, we define the system as the final product, consisting of small parts (components), but this makes no sense when working with large complex system. Thus, building up a system through an anatomy, or through its architecture is used in complex systems as a test approach during development of the system. This is a novel approach to "integration" testing, because one can call "all tests" integration tests depending how we define integration. If integration is to put two parts together to create a new part we will still have an interface to look at, and the new part is thus a "new system or component". As a contrast one talks about code level. This causes great confusion to developers, who aim to confirm that their own code works according to their understanding through execution, hence test. This means, that they must create a load module in some form of environment – often simulated, to be able to execute it.

Not all quality enhancing work is testing. Testing is by definition to test/execute a part, a file or a set of files of the code. And the only difference to "test at code level" is that the test cases are based on the know-how of how the implementation and code is (assumed to be)

created. System testing primary looks at the systems behavior, and is often synonymous with “not” understanding or caring about how the code or implementation is done.

This is the “black-box” approach. If we define the behavior as input-output, we are often talking about the functional test. If we define the behavior as characteristics, we are talking about non-functional tests, which often require analysis of different kinds. But, to make things more difficult when defining, the system is the scope of an executable as defined, this means, that the smallest component to a large complex system of system, black-box techniques are feasible. This also means that both functional and non-functional TDTs can be used at any time.

This is why levels might not be such a valuable concept after all, when it comes to TDTs, but useful when organizing the software, and dividing the scope. It seems like more level implies better control and hence better quality (more faults found). Thus the old “divide and conquer” seems to be a valid concept. In our first study on SQR, this became the consequence of our teaching of test. One phase got divided into four as a result of the improvement, to get better control over each step in the complex middleware. Instead of discussing software LEVELs it would be more feasible to discuss how the human understands different representations when it comes to discussing TDTs. This conclusion matured with each study and particularly between the different study groups.

13.2.4 Definitions of Understanding Behavior instead of Levels

In Figure 13.3 we present a competence profile for Person *Blue* that might be a system tester, in relation to a specific software system. There are also profiles for person *Green* (who probably is a developer) and person *Yellow* (who is probably a domain expert), and their specific understanding or approach to the system. Understanding the domain (so called domain experts) will only see the system through its interface e.g. a GUI, menu-system. They do not bother with how the system is organized or developed, but would see any division mainly on a conceptual level. From a testing point of view, they will always have an easy task of defining the verdict of a test

case execution, since they hold the key to how it should work in the real world, thus are instrumental to the requirements.

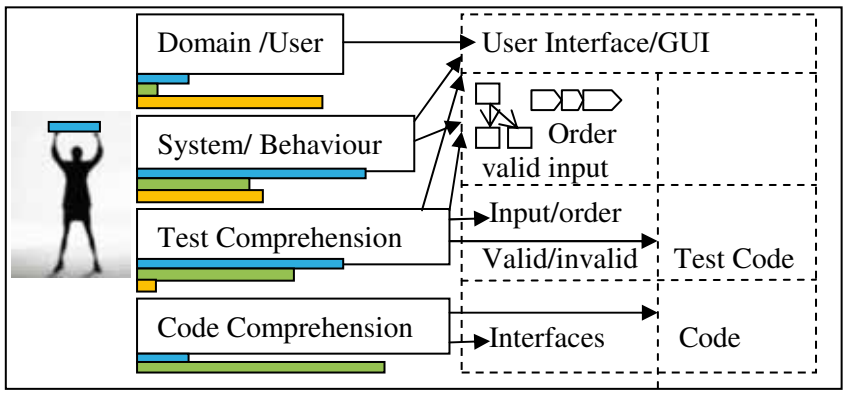


Figure 13.3 Person "blue" comprehension profile vs. system

The differentiation will be an interpretation in a judicial manner of what is "right" and "wrong" and can also be rather philosophical. In a software system, one must decide and make an interpretation that is widely understood by many (users) of the system. Domain experts and users are often (but not always) the same. A historic problem was that users had a hard time abstracting something concrete (on paper) in a manual process that suddenly got moved into a software system. One can see the similar behavior that people tend to get stuck and expect what is familiar and could be done (before), and have a harder time abstracting to "less" steps, whereas some aspects are more hidden "underneath" the visibility of the system. At the same time, it is often hard to imagine or innovate new (outside) the existing expectations, which is often the case when creating new software systems. This can pose some problems for domain experts in creating test cases and determining verdicts in test design. One solution is to give the domain experts a subject matter interpretation advantage. After all, they are the users, bearing in mind that blindly following users "processes" could results in over tedious and unnecessary complex solutions.

Behavioral understanding of a system means understanding the implementation of the domain into a particular system, including knowing which the specific limitation of input and output behavior is. Thus, how the flow of the user can be translated into modules (a model) and be organized (split into reasonably sized items).

This means that the system can be divided and organized in different manners, and the flow of solving tasks can be identified. One can identify sub-systems, particular use cases or tasks of the system, depending on how complex it is. The user interface would be any other model. Knowledge of how things depend upon each other in this division is often the key issue here, and correct organization eases the integration of the system, and limits the internal relations between parts of the system. Understanding the construction and structure of the system, this is often a primary task of configuration, architecture, anatomy and conceptual grouping of the system in a layered or tiered approach. This will focus on how parts are dependent and interact with each other, and defines typically components and their relations. Some system (often small) lacks any fundamental component of the system, and the division of these systems is often into a two level approach: the user-interface and either a database or a resulting action/computation. In particular, the main differentiation is that system behavior comprehension often focuses on how the system should work, and how it should be organized and divided (and then work). Little attention is typically spent on exploring invalid aspects of the system.

Test comprehension and code comprehension should not be an issue and be "removed", assuming that everyone working with software possess the basic skills for creating, coding and verifying software systems. This is what we in study 8 have shown not to be the case. In general, few testers or developers possess sufficient skills on test design to ensure the software is sufficiently tested. It seems to be a fact, that many persons verifying software behavior are poor in understanding software per se, e.g. code, which also becomes visible when attempting test execution automation. Therefore it is important to clarify the individual skills. In the comprehension model we describe and visualize in e.g. Figure 13.3, we put emphasis on that testers not only need to verify the positive and valid behavior, but should in addition be able to verify invalid and target negative aspects of the software, and also be able to be efficient, by automating as many aspect of the test (including design, execution, verdicts and fault location) as possible.

Subsequently, code comprehension should not be solely including creating code, without knowing if the code works in its context. Thus even if we denote code comprehension as an individual set of

comprehension, we do mean that people possesses more or less of all comprehensions in Figure 13.3.

13.2.5 Organization of TDTs

In our ambition to support the comprehension and learning of different TDTs, it seems natural to organize them in multiple ways so industry can use and select which viewpoint suits best, as well as ease the selection of techniques from different groups. The groups introduced should be seen as a new approach to look at TDTs, instead of being stuck in views that do not really aid the test of the system. We will in this chapter present the following different criteria for grouping and organizing TDTs:

1. Conventional naming, such as dividing into functional techniques with the two sub-groups: input based and structural, and then non-functional (see Table 13.1)
2. Goals or main focus of the TDT (maturity) (Table 13.2)
3. Differentiation based on observation of selected targets (Table 13.3)
4. Taxonomy Families (Table 13.4)
5. TDTs (Table 13.5-13.17)
6. Subsumes hierarchies, which identify overlap and containment of techniques

Note that it depends on the implementation and interpretation of each of these techniques if the below tables are valid. Since all techniques are possible to use within all “test” levels, such a distinction is meaningless. Not all techniques have been classified. In particular we can conclude that techniques that reside in many columns are often specified on a “too high level” and **can be viewed as a “group” of techniques**. Furthermore this also means that the target domains in a normal system are often very large with such a “technique”. It is always better to have a more narrow (better measurable) definition of a TDT, if one is to target specific failures.

Table 13.1 Functional and Non-Functional Techniques

Test Design Technique	Functional Input	Functional Structural	Non-Functional
Positive/Specified	Yes	Yes	Yes
Negative/Implicit	Yes	Yes	Yes
Equivalence Class	Yes	-	-
Boundary Value	Yes	-	-
Input Domain Combination	Yes	-	-
Permutation	Yes	Yes	-
Outside Boundary	Yes	-	Yes
Random	Yes	Yes	Yes
Fast Anti-Random	Yes	(Yes)	-
State-transition; MBT	Yes	Yes	-
Functional Coverage	-	Yes	-
Statement Coverage	-	Yes	-
Branch/Decision coverage	-	Yes	-
Basic Condition coverage	-	Yes	-
MC/DC	-	Yes	-
LCSAJ	-	Yes	-
All Definition Uses	-	Yes	-
All Predicate Uses	-	Yes	-
All Uses	-	Yes	-
Usage based	Yes	Yes	Yes
Performance	-	-	Yes
Robustness	-	-	Yes
Usability	-	-	Yes
Stability	-	-	Yes
Installability	-	-	Yes
Security	-	-	Yes
Fault-injection/ Fault-Seeding , Mutation	(Yes)	(Yes)	Yes
Search-Based tech.	Yes	Yes	Yes

Given the somewhat ad hoc nature of testing aka execution of the system, the goals of testing are many, and in many cases the only

differentiator between some techniques in Table 13.2 are related to how the techniques are either:

I. Demonstrating that a particular aspect/execution path of the software works as intended.
II. Identifying failures in execution (deviances) and contribute to the fault location
III. Collecting a sample of executions/tests to make an estimate/ measurement
IV. Checking completeness of test suites or evaluating if a test exists

Table 13.2 Different Goals of Test forms a Group

	I	II	III	IV
TDTs	Positive Search-based State-transition MBT, EP, BVA	Negative Search-based Random, FAR Permutation Security Outside Boundary, Overflow, underflow	Coverage Performance Robustness Stability Installability Maintainability Useability Security Search-based	Fault-Injection Fault-seeding Mutation Path Coverage Search-based

The groups in Table 13.2 are incomplete with regards to the set of all existing techniques, but we aim to give a general idea on how some techniques can be used for different goals. In particular one can view the use of the techniques as an indicator of the maturity of an organization. This aligns with views on "maturity" of organization proposed by e.g. Beizer [18][20]. The more mature any organization become in testing, it seems that their view of test design and their techniques develop along the above pattern (from I to IV). One has to remember that maturity could correspond to different combinations in different organizations, where advanced methods in one area can be used alongside simple method in another. Since search-based testing is present in all groups, it also shows how this technique is adaptable depending on the design of the fitness-function, and what the goal with the testing (i.e. search) is intended to be. In pursuit of understanding how you can compare TDTs, we found how important it is to be very specific with the definition of the technique.

Table 13.3 Techniques in relation of targeting behavior (A), path (B), domain (C), abilities (D)

Techniques	Behavior	Path	Domain	Abilities
Positive	Specified	Both	All	All
Negative	Unspecified	Both	All	All
Permutation	Unspecified	Path	All	Order (paths)
Outside Boundary	Both	Both	Tolerance	All
EP	Both	Input	All	Input data
BVA	Both	Input	All	Input data
State-Trans, MBT	Specified	Both	All	Order (Path), Input, Consistency, Dependency, Structure
Coverage (all)	Specified	Path	Code	Path, Structure
Fault-Injection	Depends	Any		Observability, Propagation, Robustness
FMEA	Both	Any	All	Robustness
Performance	-	-	Usage, Code	Measurement, Throughput, Time of flow
Random	Depends	Any	All	Tool
Usage techniques	Unknown	Un-known	Different variants for different comprehen sion	Measurement, Statistical behavior, Learning

Another way to differentiate techniques, presented in Table 13.3, is based on if the technique is focusing on behavior, path, input domain, user comprehension or other aspects:

A. Observing specified versus implicit and unspecified behavior
B. Observing input and/or output behavior versus observing path (organization and structure), or a combination of the two

C. Observing software based on abilities (know-how) of the user – differentiating the following comprehension levels:
    - Understanding System Domain – Business Data
    - Understanding System Behavior – Usage
    - Understanding Construction and Structure – Code
D. Observing "aspects" or an ability or a property of a particular behavior (complementing the descriptions in the other groups).

In pursuit of understanding how you can compare TDTs, we realized how important it is to be very specific with the definition of the technique. Comprehension is a big issue, and there are many nuances of how to use a technique – thus apply it – in different contexts and different systems. This also depends on your own view and knowledge level and on what abstraction level you are testing the system. It is extremely difficult to find which faults a technique finds, since test cases find the specific propagated symptom (failure) that will appear differently in different systems. Thus, connecting a technique to a code fault is not a simple straightforward matter – since the know-how on faults in the code and their pattern to propagate is still unknown.

We were also intrigued by when in the development process to apply which technique, and set out to see if there was a huge difference. To our surprise, one of our conclusions is that most TDTs can be applied at almost any level of test. In other words, you will always "execute" something and this path could be measured using supporting tools.

There exists a multitude of TDTs or TDTs for short. The number of techniques could easily be increased depending on how we define a unique test technique. We have briefly looked at the TDTs used in industry (by studying test cases), and only to a small degree were TDTs systematically applied. It seems like most techniques are named after the goal or specific domain it is applied for. We also found that by expanding existing test cases by aiming to apply one technique to unstructured test cases, improved the ability to find faults [66]. In conclusion, our experience is that testers are often not trained in testing, but on system behavior. As earlier stated, we resort to show the systems behavior is working (positive testing) Even if people are formally trained in TDTs, they easily fall back to approaching testing from a system usage viewpoint rather than applying a test technique,

since how well testers perform are seldom assessed as long as they find *some* failures.

13.3 Taxonomy of Test Design Techniques

To define TDTs, a rather elaborate scheme is needed. Not only are many names overlapping or synonymous, but they could also be used with different goals in made, even if the *practical* use or intention with the test case differs. This makes the number of variants large and since the definitions are not clear and unambiguous, it is hard to research and compare the techniques. The result of this is seen when using TDTs in practice. This results in confusion, where testers and designers have a very fuzzy picture of these techniques in understanding and applying them. The technique is obvious in one example, and really fuzzy in another. It does not help when people around the world lance their own "new" technique and approach with a different name, and that literature and papers are conflicting in interpretation. Also within companies the view is limited and own names appear, with little generality outside the specific organization. We attempt to better clarify and differentiate by providing taxonomy of TDTs aiming to both handle the historic view, books, research and practical use, as well as the aim to "standardize" the area of test design techniques (TDT). It shall be seen as an initial step to clarify and simplify the understanding and application of TDTs.

13.3.1 Test Design Technique Families

First we define the "Families" or groups in the Taxonomy (Table 13.4). This will form the foundation for the descriptive tables of individual TDTs that later follows.

In this first example in Table 13.4, the Specification (I), includes ANY written confirmed (agreed) description of the system, thus can be any or all types of requirement specifications, design specifications, use case scenarios, models of the architecture, scripts or user documentation (hopefully all are consistent). It does not include implicit aspects.

Table 13.4 Families of Test Design Techniques

	Family Name	Description/ Contains
I	Specification Based *(Positive test)*	Valid (normal) Test Invalid (negative) that are specified
II	Implicit, not specified *(Negative Test)*	Invalid Input (wrong type etc) Outside boundary (ordered set) Order, repetition etc.
III	Random Test	Any path, any input, any context
IV	Coverage-based	Measuring techniques and structures
V	Syntax (and/or) Semantic based	Usage of the programming language, declarative, interpretative, syntactic and semantic use as a means to construct the test case
VI	Search-based	Transformation of the test to a self improved search optimizing to fit a given criteria
VII	Usage-based	Based on a criteria collected when using the system to design the next test cases
VIII	Combination Techniques	Any combination of two or more techniques
i	State-transition, model based	Based on an abstraction (model) of the intended structure, but is a subgroup of positive test

This means that if e.g. aspects that normally are considered faulty input, wrong order, etc are explicitly stated (that should result in a fault message or any other specific "handling", e.g. the input is ignored until the correct valid format is given) they are considered a specification based test technique. Assumptions about behavior are not considered specified (if not written down and confirmed). Since it is specified, one can also claim that this is a valid aspect of the

system, thus many use the term “positive” testing or “normal testing” to all that is specified (even if many claim that test cases outside the boundary is always “negative” testing, or testing with invalid input. The implicit specification includes the complement of I, which is II. Together I and II cover the entire system, since these are the two main families.

Another family of techniques is Random test (III), or any aspect of random techniques. Since it basically is not defined or specified, as well as it lacks specified control of environment or in parts defined, it does not fit in uniquely into any of the two groups I or II, we define it as one own group.

Random in itself could then be divided into multiple types of random, selecting a random path, selecting a random input, selecting a random input within certain limits, selecting a random order of test cases (that would change dependencies) etc. Thus, random is often a technique best combined with other techniques. Also, techniques that possess a part of random aspects could be put here, e.g. given a notion of an initial condition, then the random would be calculated given this as an information.

It is the tradition to view coverage as an own family of techniques. These TDTs have the goal to create test cases that fulfills the coverage measurements (Family IV), and includes a lot of approaches. One can view this as the family of “structured” test, since it is based on a graph path at any level of the system. Coverage is based the factual system, and is a thus always based on the “truth” or fact. One can discuss if it is a TDT, since it assumes that the code/system is available when creating the test cases, or – as many use this technique (and also us in Study 10) as a complementary technique. This is when it seems to contribute the best.

The next traditional group is Syntax and Semantics based techniques (Family V). Traditionally, syntax has often been viewed as a group in itself, but this does not hold for the techniques that we are assigning to this group. It would be easy to make it “rule-based” but that could be the aspect to many of the Family’s (or – that would be the goal with the syntax). We do mean that these are techniques where we use the knowledge of the programming language syntactic and semantic aspects, the constructs of the language and its dynamic or interpretative binding during compilation or execution. Usually these

rules can be manipulated creating odd (wrong) or false "code" (mutants) or a fault, and the test case should be able to identify this difference. It is mostly used to make sure the test suite is complete, but one can also target specific injected faults with this technique.

The next Family (VI) is Search-based techniques, which transform the view of test design into a search. The search is then optimized to fulfill the "fitness function" or the criteria that we have as a goal for the search to find the "best" solution. This means that we can use these techniques to automatically find important input values to use to fulfill some aspect of e.g. coverage.

Family VII is the Usage based techniques. They are special and thus forming their own family, since the design of the test case is based on information captured from the execution of the existing system. The difference with the coverage Family – who is also capturing information from the existing system, is the aspect of structure in the coverage family. Here, the usage in itself becomes the information that is used to form the test cases, based on the assumption that either one are testing where the current usage/test execution is or the usage represents the areas of importance.

The final Family VIII is the Combination of techniques. We put this into an own family, and all the combinations are not yet fully explored. We are seldom talking about single input – but a combination of inputs. We are seldom viewing a test case creation only for one purpose, but for many – and parts of the process can use one technique, and another part of the process of test design can use another. This completes our main families for now, even if "sub-species" can be very large groups of techniques as well.

The reasoning we had on the coverage techniques about creating test cases to fulfill a measurements, makes us create the abstract structure, the state-transition techniques, the model-based test, the semi-structure that is not the same as the actual code but a representation of (how) the system is intended to look. This technique is using tools, and the trick is to make the model as accurate and complete as possible. This technique often starts with creating use cases, that are put together to form a model of the system. In particular, one is concerned with the flows of the system, and input often takes a secondary role. It stems from the simple Moore and Mealy machines used in automata theory. Many tools exist to ease the process and can

aid in the test case generation. The technique is helpful to humans how get a visual overview of the systems functions. Having said this, it turns out that it is easy to fall into the trap of representation, since it is the same technique, if it results in the same test case, but the format or representation differs. Since this family is a way to specify what should happen, it is in fact a sub-species of the Family I, Specified/Positive test. We use it as an example of why it is NOT an own family but we called it "i", to denote its sub-species belonging. To finalize this argument, this group could also be put into the combination technique Family IV, which also shows the difficulty of this area.

In this thesis we have mainly studied Family I and II, and III and IV in combination with other techniques (VIII). None of the Families are exhaustively studied.

13.3.2 Selecting Test Design Technique Categories

A key question for us in this research has been – given the same information, the same specification and the same system – how likely is it that applying the technique on a specific area would yield the same test case? Is the technique deterministic? Is the technique well specified? How many variants and representation is there for each technique? Is it applied similarly for different systems? Is it applied similarly at different levels?

Compared to Vegas schema [203], our investigation focuses more on the similarities and overlaps of the different techniques. She has given the schema as a selection possibility, but we have concluded that the ability of selection will not expand and improve the current way of working. Instead our taxonomy is based on industrial needs, where we have suggested an improvement model defined in the guidelines provided in Chapter 14. Since we focus on the conceptual level of the test technique, instead of going into representation differences or debating on instrumentation done in different tools, we hope to expand the body of knowledge of these techniques. Thus, it is important how a technique is interpreted, and the work by Murnane [157][158][159] to aim and define the techniques in a more adequate BNF notation limits the technique. The technique must be

unambiguous at some level of comprehension. The more we learn about specific faults and how they propagate in "any" given software, the better we can be at discussing techniques. In particular, we wanted to simplify the view by instead of looking at small nuances, finding a way to create our taxonomy of technique families, where there are more things in common than differences. Our taxonomy view is meant to aid the comprehension of the techniques. Once knowing them and beginning to apply them in one situation, one can easier see what the goal would be and learn more variants – not to be fooled by yet "a new technique", when it is an old technique applied in a different system or domain, using a different representation.

In our taxonomy we have defined the following set of categories:

i. Name
ii. Synonyms (of names)
iii. Definition
iv. Description
v. Sources of Information (references)
vi. Variants
vii. Family and Groups
viii. Subsumes hierarchy
ix. BNF definition possible, "rule-based"
x. Scalability
xi. Automation
xii. Common (known) misuses/misunderstandings of technique
xiii. Comprehensibility (learn ability) of technique
xiv. Applicability
xv. Effectiveness
xvi. Efficiency
xvii. Subjective comments

The aim is that this taxonomy should be stringent, but it does not take long until one falls into discussions about if the variation is unique enough to form a specific entry, or if it is a "sub-spices". Furthermore, we need to define similar levels. Where in one area, we discuss a group of techniques – and in the next we differentiate each into a separate entry. Nevertheless, this attempt matches the approach in this thesis. The categories are explained in detail in the next section.

13.3.3 Categories used for Test (Design) Technique Taxonomy

Creating a taxonomy for TDTs, means defining a base from which to discuss this thesis. Since it has become evident that both practitioners and researchers often lack the basic and fundamental skills of defining what a test case contains, and what use a test technique has in the process of creating a test case for a particular system, we must attempt a better definition of the basic techniques. We use and contrast our work to Vegas [203] schema, but have not ignored other sources (particularly Beizer [18] [21], Ammann/ Offutt [4], Jorgensen [125], Juristo [128], Larsson [140] and Murnane [157][158][159]).

i. Name: As the main name of a TDT we choose what seems to be the most commonly used name. This selection is subjective. See discussion under synonyms below.

ii. Synonyms: The name of a technique varies with company, industry, standard, history and habit. Therefore we are introducing the category synonyms. It might be possible that on a detailed level the synonymous techniques could have different representation, definitions etc. Where to put the limit is difficult, since the application of the technique varies from test case to test case, and from system to system. To solve this problem, we are adding the category "variant", where the meaning is allowed to be slightly altered. We are sure that this standpoint will challenge the community – since in a strict sense every "variant" with the slightest detail changed, is different, but it does in instead confuse the user trying to apply this in real life. Defining the technique (descriptively) would probably help and change the attitude to the somewhat strange suggestions of Vegas schema (see table 10.45, page 241 [203]) where black-box test and automatic test tools are both considered "a (test) technique" which most in the testing community would discard as a confusion or bewilderment. We hope with this taxonomy to better define what a technique is (the hopefully rule-based or limited way to implement a test case) that differs from the general test process, test design, usage of tools or not, what we refer to as a test approach. It might impact what and how we select e.g. data in what context we do it, but the test technique should remain the same regardless of when or where we apply it.

iii. Definition: We attempt text-based description of the technique, based on several sources.

iv: Description: For pedagogical purposes this can section can easily be expanded, whereas examples or demonstrations would be beneficial.

v. Sources of Information (references): This category has an including approach, not excluding, i.e., one could argue who owns the right of definition here. One has to consider if the seminal source written over 30 years ago is still valid or if the actual evolved view of the technique that takes it into modern system and usage should be used. Since by copyright, exactly the same writing is often prohibited without proper references or permissions, many authors (of books) take themselves liberations in the descriptions, definitions and names, resulting in a plethora of similar, but not exactly unified techniques.

vi. Variants: Because of this "plethora" of sources, it is also possible that a unique different description and interpretation is lost. It is common to see these variants, and how different interpretations of one technique often get a new name. Instead it is probably more useful to aid the understanding by clearly describing the interpretation differences as variants. A variant can have the same name – but be interpreted and applied differently. It is to our knowledge much better to be explicit when teaching techniques that are variations of interpretation.

vii. Families and Groups: To further ease the general understanding and selection of techniques we have classified each technique into our 8 Families (see table 13.4) and to groups. Families include several techniques. Each technique can only be in one family or in the combination family. Each technique has a dominating group, but can also be present in other groups:

A. Observing specified vs. implicit and unspecified behavior
B. Observing input and/or output behavior versus observing organization and structure, or a combination of the two.
C. Observing "aspects" of behavior
D. Observing software based on abilities (know-how) of the user – comprehension levels.

This classification in groups will for each test technique define both the Family (I-VII) and group (A-D) and exactly in what realm the technique is.

viii. Subsumes category: These groups are either based on a high-level separator (this includes the results from the different subsumes hierarchies as defined below in 0).

ix. BNF notation: The idea is based on work by Murnane [157][158][159] who calls this "atomic rule". In principle this category is an attempt to define the technique in a more precise manner. This also means that since the technique is "*rule-based"*, it can also be automated or semi-automated. It does limit the use and variant of the techniques, and simplifies the classification and understanding, hence is valuable for Taxonomy. We will not repeat her work, but merely identify which techniques are suitable to define in a rule-based manner.

x. Scalability: Scalability can for a TDT have different interpretations. Here we will discuss if the technique limits, or effectively uniquely define one aspect for the TDT. Often the difficulty is not to find the first occurrence of a test approach, but to be able to "size" or scale the technique for the next test case. One could claim that scalability is the ability to create new test cases.

xi. Automation: There are many aspects of automation. We conclude that test execution is always possible of all implemented test cases, even if some aspects (e.g. manual physical interventions on e.g. test environment) might be costly to automate – otherwise the test case is not a test case per definition. There are other aspects of automation. Examples of other aspects to consider are if the test case creation is possible to automate (as is) or if there are any intermediate steps in-between. For example one must always look at the system for test case evaluation, but if the goal of the test is "to add coverage" it should be possible to automatically evaluate this. Thus the implementation of the test case can pose many qualities in itself. This is partially discussed in the guidelines. In the taxonomy tables, only the test case creation automation or aspects thereof has been considered.

xii. Common known misuses/misunderstandings of techniques: Common known mistakes for this specific technique. Knowing this aids comprehensibility, applicability and understanding. Our study in Chapter 11, but also in the studies in chapters 7-10 and 12, identifies some common patterns that should be avoided.

xiii. Comprehensibility: This can be expanded into the learnability of the technique. But comprehending the technique theoretically has shown to be totally different from applying it in reality, i.e., transforming the knowledge into useful test cases. We have defined this to four levels: A-D (see Section 13.2.4).

xiv. Applicability: Ability to find the location in the software where to apply the test case; as defined in Chapter 2.

xv. Effectiveness: As defined in Chapter 2. Main reasons are to (1.) add coverage and (2.) find failures (faults/defects).

xvi. Efficiency: As defined in Chapter 2. The overall time from creation of a test case, to a completed evaluation of the test case,

xvii. Subjective comments: To ease the usage of these techniques, we take the liberty to add subjective comments regarding our current experience with the techniques. This is a place for more experience-based (and not scientifically shown) remarks, typically based on limited or undocumented observations.

13.4 Test Design Techniques Tables

The following tables describe the specific families of techniques, and in some cases, specific sub-species. The key here is that even if it is a family – one can use this "family-description" and directly derive test cases based on its rather "wide" description, i.e., even if there is a technique that fits completely "within" the "family" of techniques as a sub-set, it still inherits all of the family aspects.

Table 13.5 Specification based test

Name	Specification based test
Synonyms	Positive test, Requirement (based) test, Conformance test, Normal test, Valid Test
Definition	All test cases are based on an explicit statement/ specification/abstraction, of how the system should work (or should not behave).
Description	The explicit statement should be in written form. This test could require few or many steps of execution, depending on how the system is constructed and works. One or many input values could be used. The limitations of the techniques are related to the limitations of the system. Any input is allowed, provided it fulfills the explicit statement, as well as any path. Also "fault-scenarios" explicitly stated are within this technique. The more well-defined (formal) the specification is, the easier the validation is. If it is not formal, it will resume to verification.
Sources	[163] [133][134][167][194], Attack 2[211]
Variants	Default Test – is both much more selective, and a subset of the technique – since it indicates that one or only a few inputs is enough to validate the test case. This is also often assumed the same as a "normal" test (the normal case), the typical and most used test case (even if default is not defined). Valid Input Test – this is also a subset – since it limits the test case to the "valid" set. It is possible to have positive or specified tests require invalid input.
Family & Group	This is Family I and consists of "any" type of representation that is described, thus all groups.
Subsumes hierarchy	1a
BNF definition possible, "rule-based"	Since the technique is too broadly defined, a rule does not aid the test case creation, thus, all TC as defined based on the specification is true. The problem would be in formally defining and

	capturing the system description.
Scalability	This technique works on any system that you can create a test case for, that has any explicit statement, and also at all levels of test.
Automation	If test cases can be automatically generated depends on the formalism used of describing the specification,
Common (known) misuses/ misunderstandings	People tend to test the obvious with this technique. Many interpret "positive" as the technique only allows for "positive" numbers, which is an imposed limitation. Or similarly, assume only numbers are to be tested for an input requiring a number. If fault scenarios are explicitly stated, they are also "positive" test cases. All aspects outside the specification or implicitly stated is negative tests.
Comprehensibility (learnability)	This is intuitive and simple to learn
Applicability	This is easy to apply. Shows what "should" work.
Effectiveness	The likelihood of finding faults seems to be related to the complexity of implementing and interpreting the explicit statement
Efficiency	Since it is specified, thus easy and intuitive – it is as efficient as the explicit statement it is attempting to show.
Subjective comments	Quality of test is related to the know-how of the system. This technique seems to be the most common, and most easy technique to learn and deploy, and is in industry the most commonly used.

Table 13.6 Negative Test

Name	Negative Test
Synonyms	Unspecified Behavior, Crash tests, Breaking software, Attacking, Unusual tests, Outside norms test, Robustness test, Abnormal tests, tolerance tests, invalid tests, Challenge tests, malignant tests
Definition	All test cases are aimed at challenging the system outside what explicitly has been stated in the specification, thus implicit knowledge and un-described behavior. It is the inverse of Specified test.
Description	This means deliberately trying to find ways to execute the system, e.g. give input that is not explicitly allowed or do things in a wrong order (see permutation). This is a "group" in itself.
Sources	[167] [211] [142] [217]
Variants	Includes Attack 1a, 1b and Attack 2, 3, 4, thus specific approach/sub-definitions as different techniques (see Chapter 10).
Family & Group	Family II, all categories of groups.
Subsumes hierarchy	-
BNF definition possible, "rule-based"	Since the technique is so "open", it cannot be uniquely rule-based.
Scalability	This technique works on any system that you can create a test case for, that has any explicit statements, thus at all levels of test.
Automation	We have not been able to define these tests in any generic rule-based, thus automatable way. If the explicitly defined input can be extracted from the system, one can pose some possible schemes of permutation of order OR using the ascii-table "rules" to generate a set of "negative" tests.
Common	This has nothing to do with negative input (though,

(known) misuses/ misunderstandings	much software only uses positive numbers). Make sure you have correct assumptions on the system.
Comprehensibility (learn ability)	The more you know about software (programming), faults, and the system, the easier the technique is to comprehend.
Applicability	For many systems this is not easily applicable.
Effectiveness	The likelihood of finding faults seems higher than for most other techniques.
Efficiency	This technique is fast to apply in unspecified systems, and slow in well-specified systems. It is very relative to the test object, and how difficult (and important) creating test cases are for this area.
Subjective comments	Any attempt to address new paths that is not specified, are more likely to succeed in its fault finding ability, since it is addressing an area probably not tested before, yet, its definition gives little help in creating the test case and a lot of variants are possible.

Table 13.7 Outside Limits Testing

Name	Outside Limits Testing
Synonyms	Outside boundaries test, extreme testing, extreme value test, stress testing. Crash tests, Breaking software, Attacking, Unusual tests, Outside norms test, Robustness test, Abnormal tests, tolerance tests, invalid tests, exhaustion test, over the limits
Definition	Test cases are designed in such as way that the input values are outside the specified/allowed boundaries for them.
Description	Limits could be of any type or sort (resource exhausting, input, number of input given for a field, outside the index etc). One could also be attempting to be outside "machine maxima" or "machine minima".
Sources	Attack 1.1, 1.2, 1.3, 3 and 4 in Whittaker [211] Confirmed in Study 4 (Chapter 6) and Chapter 8.
Variants	Resource exhaustion, stack, heap built, … A variant is also only using this technique as it is strongly related to boundary value analysis technique (see table 13.x). In that case it is used to only identify the input outside the defined limits.
Family & Group	Family II, Negative testing, Group A (can be both specified and targets unspecified behavior) and group B particular targeting input, but could also be used into exhaustion in Group C.
Subsumes hierarchy	-
BNF definition possible, "rule-based"	Since the technique is so "open", it cannot be uniquely rule-based, but on some levels input limits are defined, and thus rules can be created.
Scalability	This one aspect of the technique is simple to use on code, level, where input are often well defined. The interesting problems thus though arise on the integrated system and targeting system aspects. Exhaustion outside the "limit" of a buffer, e.g.

	heap is much harder to define.
Automation	Using tools to generate data is often a necessity to get the speed of exhausting the limits of a system.
Common (known) misuses	People often mistake this technique with Boundary Value testing
Comprehensibility (learn ability)	The more you know about software (programs), faults, and the system, the easier it is.
Applicability	This technique assumes there are ways to establish these limits, which are often a trial/error attempts that might be time consuming. If limits are known, it is often easer to address outside its known boundaries.
Effectiveness	The likelihood of finding faults seems to higher than most other techniques, since taking care of this often means extra code and control code.
Efficiency	If boundaries are known, it is efficient, otherwise trial and error is very time consuming
Subjective comments	This technique is often in attempt of malicious behavior to crash systems. Accidents are also a reason (leaning on the keyboard – getting a lot of input) in a category that cannot handle too many characters, or plain overloading buffers with input challenging the dimensioning of the system.

Table 13.8 Equivalence Partitioning

Name	Equivalence Partitioning
Synonyms	Equivalence Class, Category Partitioning, Input Domain Class, Input domain partitioning, Property based testing
Definition	The domains of input data values are partitioned into disjoint subsets (classes). The tester "assumes" that all values from a class are treated the same by the program/system and hence only representative inputs from each class need to test.
Description	For any given input, there is at least one class (all input possible). But more often, there is at least one partition for all values that are valid, and one partition for all invalid values in the input domain. More often there are several sets or classes of inputs that are handled different by the program.
Sources	[19][163] [127][171][182]
Variants	Assuming this only handles input being integers. Property based testing implicitly include that you specify the valid and invalid states (as in a rule based BNF) (the property of input).
Family & Group	Family I, Targets Group A Specified and B Input
Subsumes hierarchy	1a
BNF definition possible, "rule-based" "atomic rule"	On a code level it is possible to determine uniquely the partitions, and depending on the system, some of these partitions might aggregate intact to the user interface, but this is not for certain. Thus at a specification level, it should be possible to define – hence, it should be possible to test for it. Thus, if a nominal scale exists, the classes can be assigned accordingly, and be derived from "branching" information to the test case.
Scalability	This technique is scalable if the input is scalable in the system under test

Automation	The classes are somewhat difficult to automatically create, but they can be generated bound the scope of statistical analysis tools – or if the specification is defined in a way that makes it possible to generate the input domain automatically. Once the sets/classes/partitions are defined – it is very simple to generate data from the different categories – and thus automate the test case generation.
Common (known) misuses/ misunderstandings	The assumption is often to limit the technique to numbers, integers. People often fail to see the use of this technique for any type of set, for any type of input.
Comprehensibility (learn ability)	This seems to be related with the know-how of math, in particular the relations of "equivalence class" and basic set-theory.
Applicability	When the sets are defined, it is easy to apply. If input domain analysis is not done, the indication is that this is difficult to apply.
Effectiveness	This technique seems to aid in defining good data (classes) – and thus add coverage.
Efficiency	Input domain analysis takes some time, but then the technique is straight forward.
Subjective comments	Quality of test is related to know how of the system input domain possibilities, and making reasonable assumptions on the higher levels. On code level, tools should aid in supporting this type of testing. This technique is not used in relation to its merit.

Table 13.9 Boundary Value Analysis

Name	Boundary Value Analysis
Synonyms	Boundary Testing, Boundary Value Testing, Boundary Analysis
Definition	For each given input, a set of boundary values are identified wherever feasible. The test cases are generated for each such boundary value as well as its two close neighboring values (one smaller and one larger than the boundary value).
Description	Depending on type of boundaries there are often many more values that could be targets, depending on the transformation of the code into different formats, byte-order etc.
Sources	[19] [127] [142][161][163][182] Attack 1.4 [211]
Variants	The most common variant is two-value boundary testing. Meaning you test on "each side" of the boundary value specified. (>, <) . This is best used for manual testing. Automated testing should always aim to test three-value testing. One can use this technique if defining a countable ordered (sub-) set.
Family & Group	Family I, Group Specified (A) and Input (B), Functional, input
Subsumes hierarchy	1a
BNF definition possible, "rule-based"	This technique is straight forward on code level (identifying the following relationships: >, >=, =, <= and < as boundaries). Implicitly this as the branching based integers.
Scalability	This technique is scalable to all levels of test
Automation	It is possible to automatically generate the test input on a code level.
Common (known) misuses/	Many (60%?) actually test **the** boundary as the only input. This is not sufficient (see study 4). Many also mix what is a scale that has boundaries.

misunderstandings	Eg. Testing “dates” with this technique is not a countable ordered set, though there is a set with high dependencies (different lengths depending on month, year, leap year etc).
Comprehensibility (learn ability)	This seems to be related with the know-how of math, in particular the relations of “equivalence class” and basic set-theory.
Applicability	Lower (less test cases can be generated) than from Equivalence Partitioning. When the boundary is identified it is simple, and particularly on code level, where the limits might be more well-defined. Boundaries are often very limited on a system level, and not so easy to identify, depending on the system under test. For economical systems (handling integers, numbers) this technique is common-place and important. Thus, it is a very limited applicability in most systems for this technique.
Effectiveness	When applicable, this technique seems to target important problems fast, and provide good data and add coverage.
Efficiency	The first and obvious boundaries are often easier to identify. Other types of boundaries (used in conjunctions) are often rather difficult. It is very fast when boundaries are known.
Subjective comments	This is the technique to start with, adding simple and important fault-finding test cases. Since it is limited, it will not be sufficient.

Table 13.10 State Based Test

Name	State Based Tests
Synonyms	State Transition test, State Chart test, Model Based test, Cause-Effect graphing, Finite state machines (FSM), State-machines, Automata, Transition and graph testing, Use-case graphs, Use-case models
Definition	The basic requirement for state-based testing is to use a state-machine representation of the system, modeled with states, and transitions between states with associated actions to identify the set of test cases.
Description	The model can be described and transformed in numerous of ways. Cause-effect, a Table, a start and an action etc. The technique is a more "formalized" way that has direct overlap with positive testing, thus, what is defined and specified can be modeled. One individual use-case is often not considered a model, but is one "path" through the model. If all use-cases are combined – it will represent the "system" or some part of the system. One can define a transition for one or a set of inputs. Using equivalence class techniques one can further define different transition for different input. There often exists a "starting" state. And one can also define "time-out" as a factor for changing state. Furthermore – this technique is possible to represent in numerous ways, given different formats and representations.
Sources	[217]
Variants	End-to-End testing which is only one flow/special case that can be defined like this.
Family & Group	Family I, Group A, Specified, Group B Both input and path
Subsumes	1b

hierarchy	
BNF definition possible, "rule-based"	Applying the technique is human (from problem to specification/model). Once the model exists, test case generation is possible. If a semi-formal specification of the system exists, creating a similar test model is often straight forward (often the system model can be reused and complemented).
Scalability	This technique is scalable.
Automation	Applying the technique is human (from problem to specification/model). Once the model exists, test case generation is possible- thus is automate.
Common (known) misuses/ misunderstandings	Many people tend only to model, describe and specify aspects of the system that should work. Tools also put limitations on how to use the technique (e.g. do not allow looping; do not allow some forms of input etc). Thus, models are mostly focused on "valid" input, and have problems of defining negative tests. Note that the model states and transitions are only as good as the tester defining it knows the system.
Comprehensibility (learn ability)	This seems to be a very intuitive way to "picture" the system that seems easy to learn and create test cases from at any level.
Applicability	The first levels of simple positive test cases are often easy to model. It is much harder to find negative tests, holes in the model (missing areas), otherwise tools often make test case generation relatively simple.
Effectiveness	This technique (because of today's status of the tools) seems to focus more on getting specific paths and uses-cases implemented, and thus the normal case). Inconsistencies can be found during modeling, and good transition coverage can be generated. Thus, this does not imply that it finds many faults.
Efficiency	Efficiency of the technique is tool related, and

	thus relate to how well the tester know its tool and defining the test cases. It often requires extra work to define evaluation of the test case, since the tool often only support the execution, but does not (always) construct complete test cases, i.e. including evaluation information.
Subjective comments	The quality of the test specification supplied in these systems often becomes better than manual text. This is an emerging field that is related to the progress of the tools. Graphic support seems to aid beginners in understanding how a system works.

Table 13.11 Search based test

Name	Search Based test
Synonyms	Genetic test, Evolutionary test
Definition	In search based testing, a meta heuristic search technique with a fitness function is used to select test data which are 'best-fit' in the near optimal sense with regards to a specific criterion.
Description	This is a Family of techniques. This criterion could e.g. be that the execution attempts to identify an aspect that adds coverage, and adds test cases that contribute to improved coverage. The best you can do is "Pareto-front".
Sources	[29][150][206]
Variants	Hill-climbing, Local maxima/minima, Simulated annealing, Ant-hill, Tabu-search, Cross-over, multi-objective searches
Family & Group	Family VI Search-based. Any group, C, aspect would fit. Note that this is common use in combination.
Subsumes hierarchy	1d
BNF definition possible, "rule-based"	Yes. This can be formally defined. Evaluation criteria must be formally defined.
Scalability	Unknown
Automation	Possible to generate test cases, but there are many aspects that must be specified and set up for this to happen.
Common (known) misuses/ misunderstandings	The "fitness function" can make the technique into any (other technique).
Comprehensibility (learn ability)	This is a difficult technique to comprehend initially, and assumes many skills.
Applicability	It is currently only partly explored in research.

Effectiveness	Hill-climbing seems to be more effective – but it seems to be more relevant to formulate the search and fitness function correctly for the different goals.
Efficiency	For Industry this area shows promise for some optimization tasks, but it is still a bit uncertain as a genuine test method, and more research is needed.
Subjective comments	Very interesting for future work, especially in combination with other techniques.

Table 13.12 Coverage based test

Name	Coverage based test
Synonyms	Code coverage test, execution measurements
Definition	A test case is created with the aim to (execute) cover a specific execution path in the code, OR a specific aspect of data from an input domain (condition or usage). Any graph or path or structure based on the existing system, forming a measurement, can have an appropriate coverage by a test. Based on a coverage criteria (e.g., the set of paths or statements of a program), defining a unique set of elements such that no pair of selected tests corresponds to the same unique element.
Description	This is a group of techniques.
Sources	[4] [30] [221]
Variants	Many, often dependent on tool, language etc.
Family & Group	Family IV Coverage, and Group A specified, and Group B path/graph.
Subsumes hierarchy	See [30]
BNF definition possible, "rule-based"	In principle possible.
Scalability	Have limitations, since for larger systems instrumentation might cause serious problems (this is system dependent). For data-flow coverage, there are often an "explosion" of data that causes problems
Automation	This is not used without tools to aid the coverage measurements/instrumentation.
Common (known) misuses/ misunderstandings	Since coverage is possible to complete at the measurement level "100%" it is common to confuse the coverage metrics with "completed test".

Comprehensibility (learn ability)	Code comprehension. Easy to learn for simple coverage metrics. Tool-dependent, so details are often hidden from user how the system is instrumenting the code.
Applicability	Applicable for all systems and languages
Effectiveness	Will find faults, especially in the high-end of the coverage (the more hard to reach executions)
Efficiency	It is time-consuming, but definitely worth the effort it if done in the right order (functional tests first, then complement with coverage tests).
Subjective comments	Coverage is clearly defined for most cases, but instrumented differently for different languages, compilers etc, thus the result (measurement number) might vary within approx 10 % based on tool. It is more difficult to yield higher % the more aggregated (integrated) the system is. It is easier to get high coverage on the smaller units/blocks or modules of the system. Data-flow coverage is not sufficiently explored for industry.

Table 13.13 Random Test

Name	Random test
Synonyms	Random Input Test, Random Execution Test
Definition	The creation of test cases is based on a mathematical function that creates a random number which is then used to either select a random input (from a defined input domain) or a random execution path (from a random set of test cases).
Description	The Random selection based on mathematical sound principles must be used in the technique.
Sources	[152] [204][220]
Variants	-
Family & Group	Family III Random, Group: C “any aspect”.
Subsumes hierarchy	-
BNF definition possible, “rule-based”	Depends, within a specific set up, this is possible.
Scalability	High scalability
Automation	The technique benefits from a fully automated set up where semi-automation (only the random generation) is not satisfactory efficient
Common (known) misuses/ misunderstandings	Ad Hoc testing is often confused with random, since it is a human “choosing at random”. There is support ([194], Chapter 7, 8, 11) that humans do not choose at random, but are coherent in their test case, input etc selection. Note that many “random” functions are repetitive in their pattern.
Comprehensibility (learn ability)	It is relative easy to understand and learn the technique. It is more difficult to set it up in a useful manner.
Applicability	High, this is
Effectiveness	Low, this will create a lot of “wasteful” test cases

Efficiency	If set up automatically, random input has shown useful for input data dependent software – especially in combination with other techniques.
Subjective comments	

Table 13.14 Fast Anti Random

Name	Fast Anti-Random (FAR)
Synonyms	-
Definition	This is a random technique but with a "dependency". Since the first "value" is selected random, the next "value" is selected according to an algorithm at as far distance as possible.
Description	The technique selects from an Input domain and calculates "distance": The technique can also be used to select a "path" and then calculate a "distance" to the next path.
Sources	[36]
Variants	-
Family & Group	Family III, Random
Subsumes hierarchy	-
BNF definition possible, "rule-based"	Algorithmic, but initial random is a function that should be variable. Meaning, it is rule-based.
Scalability	Scalable
Automation	Should be automated
Common (known) misuses/ misunderstandings	-
Comprehensibility (learn ability)	Seems intuitive easy, but have areas that are difficult, should be solved by the algorithms.
Applicability	Not evaluated
Effectiveness	Not evaluated
Efficiency	Not evaluated
Subjective comments	Not enough used – since Random seems not to be common in Industry.

Table 13.15 Repeated tests

Name	Repeated test, repeating series of inputs or executions
Synonyms	Memory exhaustions, limits exhaustions, resource testing
Definition	Repeating a series of inputs and executions over and over
Description	Aims to repeat a series of inputs and or a series of executions over and over to try and exhaust the system.
Sources	Attack 6 [211]
Variants	Automated test case repetitions, exhaustion test, (some variants of stress test).
Family & Group	Family II Negative Test targeting unspecific behavior in Group A and unspecific behavior, and B both Input and path Family I (when used e.g. in characteristics measurements e.g. performance)
Subsumes hierarchy	
BNF definition possible, "rule-based"	For every x, where x is a test case, input, or describes a series of execution steps, repeat x until finding a fault or a very large number is achieved of the repetition.
Scalability	Yes
Automation	Yes
Common (known) misuses/ misunderstandings	No
Comprehensibility (learn ability)	Simple
Applicability	High
Effectiveness	Depending on system properties, but yes in most

	systems this is valuable technique
Efficiency	If test case is automated, it is effective.
Subjective comments	No explored sufficiently in Industry

Table 13.16 Permutation

Name	Permutation test
Synonyms	Order based test, order requiring, order variation, dependability test, arbitrary ordered usage
Definition	Describe that test steps, test cases, or complete test suites can be permutated thus done (executing) in arbitrary order yielding the same result.
Description	A system does often have many “entry points” of execution or use, and can be addressed in different ways. Sometimes specific order of events is required; then permutation within that order will become a negative test approach. Showing that something works, starting from different positions could also be questioned if it is a true.
Sources	[19][163]
Variants	There are two ways of making permutation. The easy is just to swap between test cases, since they are forming a unit, this is possible, and easy to implement. Secondly, the next variant is to change steps within a test case – this takes much more work, and is not simple and intuitive.
Family & Group	Most common is Family II Negative test Group: Paths and order of events if not strictly sequential. But possible to use as positive test in context of repeating different permutated sequences for characteristic measurements (see repeated tests).
Subsumes hierarchy	-
BNF definition possible, “rule-based”	Could be possible
Scalability	Yes
Automation	Yes - advisable
Common (known) misuses/	-

misunderstandings	
Comprehensibility (learn ability)	Easy on a “random” level, very difficult if a thorough analysis are to take place what is (or should be allowed) and not
Applicability	Most systems that are strictly sequential
Effectiveness	Depends on how well the system is specified (could be for some domains).
Efficiency	If automated yes.
Subjective comments	Somewhat explored, but need more data in research to fully enjoy. The simple variant is an easy way to utilize existing test cases if automated.

Table 13.17 Combinatory test

Name	Combinatory test
Synonyms	Call-pair, pair-wise, dependency tests, input combination test, matrix testing, combining test, factorial test
Definition	Combining one input depending (or following) one or more inputs.
Description	Aims to combine two (or more) input domains in a matrix and thus create test cases that cover a specific combination. Is a subset of Combination Test.
Sources	Attack 5 [211] [214]
Variants	Configuration tests (which is a special case), where the actual set up have a series of different combinations of hardware, parameter settings or situations that needs to be tested in combinations.
Family & Group	Family VIII, Combination test, but are using specified Input domains, Group A, and Input (Group B) (Input analysis) and could be seen as belonging strictly to Family I.
Subsumes hierarchy	Input analysis
BNF definition possible, "rule-based"	Yes (but difficult with large input domains).
Scalability	Yes (but difficult)
Automation	Yes
Common (known) misuses/ misunderstandings	One often assumes that simplifying a "many" combination (multi-objective) problem into a simple test "unique-pair" problem is sufficient.
Comprehensibility (learn ability)	Requires some math skills (factorial)
Applicability	High

Effectiveness	Depending on system properties, but yes in most systems this is valuable technique, and can cause serious problems.
Efficiency	With large input, this is a simplifying step, making the selection order simpler increasing the risk of fault.
Subjective comments	This technique is often lower priority, since it requires much more time and systematic approaches. It could be said it aids the complexity for most systems.

These tables are an incomplete set of tables, but also the presented tables lack information to fully describe the techniques. There is a lack of completed research in this area. We have selected the above techniques since they represent some of the currently most important techniques. We have presented the TDTs both in their most generic view, but a few as sub-species. The selection was made to demonstrate that our approach is feasible for defining taxonomy of different techniques. We do admit that this is just an ad hoc selection very biased to the techniques available and known to the researcher who is primarily interested in exploring each of them from an industrial standpoint.

13.5 Comments on TDTs Related to our Studies

In our studies we have looked at Specified test, Equivalence partitioning and Boundary Values. We have also (in Chapter 10) used randomized selection within each input domain (that is rather narrowly specified). Finally we complemented this with the use of just a few coverage techniques. In addition we explored the Negative TDTs Chapter 9. Our studies aimed to target both positive and negative testing. The most common mistake using equivalence partitioning is that the input analysis is incomplete, and subgroups are not thoroughly defined (there might be many different subgroups within the partitioning – for which the system reacts differently. Since this techniques assumes that all data in a group is treated the same by

the system, it should be arbitrary which value is chosen from a particular group whilst testing. Since groups might contain large inputs, it is very common that the assumptions are wrong. Even if an input (valid) might be "all positive integers" it might be possible to find a very large number that belongs to that group that is invalid for the system to handle.

The most common problem is that many assume equivalence partitioning only to be valid for numbers, or even limiting it to integers, as well as ordered sets. This is one out of many interpretations that might not be entirely true. Using the entire ASCII table as possible input will in fact often invoke the system and create a lot of "negative" or "invalid" subclasses of input. We learned in the process of exploring negative test case, that much more character sets, representations based on different operating system uses etc impacts the domain.

One would easily draw the conclusion that Boundary Value Analysis is a sub-set of Equivalence partitioning, but are in many aspect limiting the type of input to numerical ordered sets, where a clear "boundary" between valid and invalid is countable. In code construction, the usage of text is often translated to test the ASCII – which is a numbered sequence. And here we often see mistakes done on the usage of the technique. Another mistakes area seems to be in the usage of dates, whereas the days are consecutive in order, the Gregorian calendar are not a typical ordered numerical set, since after day 30 it is not always day number 31, and after day 31 is not always day 32. In fact these are sets of data, with some special circumstances – depending on a set of rules, like leap year etc. Often a series of negative and positive test cases are attempted to identify an existing boundary in a number field, if the limits are not known. This is not applying boundary value analysis technique – and is a common misconception. Knowing the actual boundary, and giving appropriate 2(3) input values, is applying the boundary value technique correctly.

13.6 Subsumes Hierarchy of TDTs

This section provides a new way of presenting TDTs, but focusing on how different techniques "cover" or subsumes other techniques. The idea is to define a hierarchy, which from top-down defines

successively smaller domains corresponding to more specific selection sub-domains. This idea supports how we group techniques, and why we find similar type of faults in similar groups. A more refined division is that the "lower" in the hierarchy are the most effective to reveal faults, and gives fewer possibilities to select the execution.

13.6.1 Selection of Input Domain

The first subsumes group, is based on TDTs that specifically address the input domain. It is selecting one (or more) test cases containing one input selection.

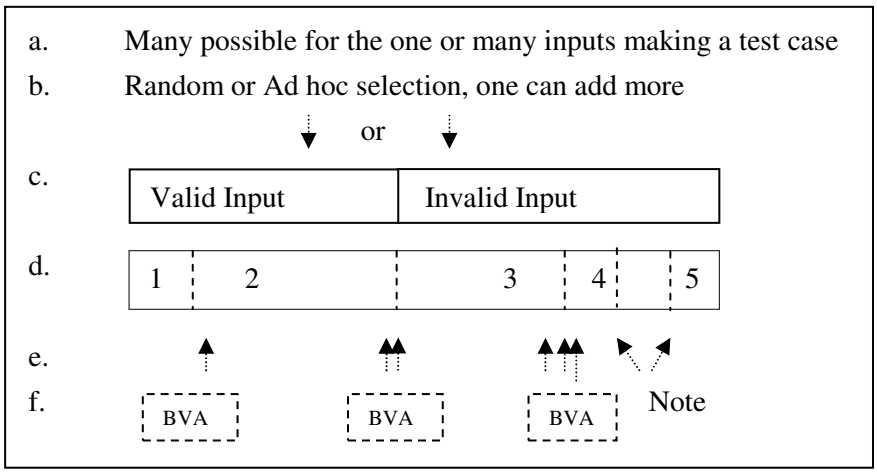


Figure 13.4 Subsumes input domain

For selection of input one can partition the input domain space into valid and invalid inputs**.** Consider Figure 13.4, in which the entire input domain is described in a. (any character set, any type representation, etc). In a. any value given will have a corresponding execution. But if one would generate input by any form of random generator from the entire input space, or it could be an ad hoc selection, we are now addressing b. When an ad hoc selection is performed by a user, there is evidence that the selection is almost always selected from the valid input domain space []. The input domain set can then be narrowed down to more specific areas, such as in c., where valid are numbers in a specific character set, type and

representation and specifically the format given for this particular execution event. Defining input as valid and invalid (c) invokes the simplest level if division of the input space. The next level (d.) is to divide the input space into valid equivalence classes. This means that an evaluation of the input domain is taken one step further, and the internal actions of the execution or the program behavior is taken into account.

For the next level, e., Assume that the valid inputs are the values -5 to 0 (which result in one action) and that input from 1 to 42 result in another action. Values below -5, above 42, and any other character in the ASCII table, that can be given as a written input are invalid for this specific case. In this case, the boundary value analysis technique can be used. In fact, assuming the input only accepts a 2-character input delimits the input space in this example. Boundary values are -6, -5, -4, (for the first ordinal scale limit – 5), and then -1, 0, 1, and finally 41, 42, 43, but the transition to e.g. Aa is not a countable difference (if not the original ASCII value is used, but that cannot be assumed) and looking at the problem, at least one group will be -9..-6 (since the minus will take one character) and another group would be 413..99. Here it is assumed that the software would have different actions depending on if the value is below -5, or above 42 – forming two different invalid equivalence partitioning. Furthermore, one might want to explore "outside" the input space, e.g. the boundary between 99, 100 and -9 and -10 utilizing the Boundary Value Analysis Method. To further explore, we might want to use the variant of the Boundary Value Analysis method, and explore the same values, but in different representations, f. For all other character one could maybe identify a series of groups (e.g. between 4^{th} class and the 5^{th} class, and similar between the 5^{th} and 6^{th} class) that there exists no particular "boundary" since the actions are not grouped based on ASCII, but grouped based on the software execution paths or particular action that should take place. One example could be that all characters (Ctrl-x, Esc etc) would be in group 6 and be a communication with the operation system, effectively terminating the program. One could call them "invalid" input for this particular program. In another instance, the same values could be considered "valid" input. That is after all a "human" decision on how to regard the program. The example (classes) in d. is by no means complete or really accurate, but just written as one example on the reasoning, where one can consider the

equivalence partitioning (classes) be on a more coarse (and thus less specific) level than the boundary value analysis technique, that furthermore require an ordered (enumerated) set. Note that several variants of interpretations of Equivalence classes and equivalence partitioning exist (discussed in Table 13.). Note that Boundary Value Analysis is a "generic" model that can apply to any type of ordered formats, not only ASCII values representing characters. Other boundaries could for example be byte order (looking at a string of O´s and 1´s) that could have impact of the quality. This example was the area we studied the most in our set of TDTs.

13.6.2 Specified subspecies: State based

The state based techniques are another subspecies of Positive techniques. One can follow the states based on different abstraction levels, where at one level the state is a singularity, and at the next level, one can simplify a combination of many single states, into one new state (going from bottom up). From top-down, one can go from a wide variety of test cases fulfilling the higher level, but making the selection more and more specific into finally reaching the specific (section of) a code-path executed with the specific input.

a. Specified input (Family I)

b. State based techniques (i) which is the group that includes State Transition Model = State Chart Model = Model Based Testing (different formats possible: table, graph etc) (all use cases)

c. (single) Use-case, several transitions

d. Single transition – Single action

Figure 13.5 State based subsumes hierarchy

As stated earlier the first "level" (a) is the specified input wherein this sub-species group is included. Then e start with this state-based techniques (b), that includes any state transition technique, the state chart technique and model based testing, that all are different representations of a "model" or abstraction of how a system works or is specified. The model could be a model of models, thus, this can both be used to simplify interactions and have several abstraction

layers. Based on what format we select, the model subsumes a specific single use case (one could easily see a model being the result of several use-cases put together to form a representation of the system). Whereas a single use case (c) that is compromised of several actions or transitions, subsumes a single transition or a single action (d).

The interesting part of this is that none of these techniques is "complete" in the aspect of ALL paths or ALL inputs. Thus, it is very common to create use cases based on requirements (valid) and then create a model of the system that implicitly only tests positive and valid input (but neglects all but a few invalid inputs). This is probably because most modeling of a system does only describe what should work and lacks the description of what should NOT work. Improving with this aspect, would contribute to better test of a system.

Understanding granularity and abstraction levels is the core knowledge when it comes to TDTs, where the single action or transition should represent a unique sequence, a statement or "path" in the software (code). Determining boundaries for what is a "single transition" might be the difficulty of the system, but might also be obvious, but should be held together with the type of system under test.

13.6.3 Family of Coverage Subsumes

We have not spent time on defining subsumes hierarchies, which are partly described in [BS7925-1]. In particular, we can find important work made by Juristo et al in e.g. [129] highlighting the important findings of Weyuker [208][209] for data-flow coverage, where she talks about the includes relationship, instead of subsumes. This work has been instrumental and inspired the view and creation of Taxonomy families.

13.7 Contrasting with Related Work

In this section we describe some of the main related work to this attempt of a new taxonomy. There is a plethora of knowledge in the area, but we have restrained ourselves to original work by some

specific scholars. First, we present a recent thorough empirical work which is the only work with a goal similar to ours, Vegas [203] supported by Basili [201]. Then we compare ourselves with more recent thorough book by the Ammann and Offutt [4]. The overall influence of Beizer [18][21] cannot be ignored, as well as the small, but important insight by Murnane [157][158][159] approach to make techniques bound by a syntactic definition. To be useful the rigidity of defining techniques still needs much more work.

S. Vegas in her PhD [203], aims to assist the testers/developers select what TDTs they should use. Her theory is working under the assumption that all techniques stated (with different names) are in fact "different". Our view on this differs greatly, since many techniques result in similar test cases.

Variation could be due to the fuzziness of definitions that allow for many interpretations, but a similar application would result in similar test cases. This is because the techniques are applied by humans, thus individual preferences occur.

Secondly, we also find that the same technique can be given a different unique "name" and consequently be viewed as different, when in fact it is the same technique. This might be depending on reasons that the technique comes from a different context, might have a different representation, or have been used on a specific type of system. This has caused the test technique explosion that is currently dominating the discussion.

Vegas thoroughly investigated testers' and experts' "opinions" on these TDTs. Many so called "experts" have never tested a great variety of different types of software in practice, and if so – it is often similar types of systems. It is no wonder that there is still room for renaming the same phenomena over and over. Even if the selection methods stated by Vegas are empirically sound, it seems like many people obviously fall into the same traps of understanding the techniques at depth, thus, the basic assumption that the TDTs are different is – in our opinion – just wrong. We have run two different projects (see chapters 9 and 10) to see if the testers did discover any form of overlap in the techniques, but failed to do so. So it is an easy deception.

Vegas has not critically challenged the approaches and similarities with the different "techniques", and does for example differentiate

"decision tables" and "state-transition-testing" as different methods, where the resulting test case is the same, and the difference lies in how the test technique is represented (in a table, or as a graph). She furthermore claims that "black box testing" differs from "functional testing" and she is also making the boundary value technique an explicit white-box technique, although the technique could be used both as a black-box and white-box technique (even when taking different definitions of white- and black-box into consideration).

In fact, Vegas implicitly define some characteristics of TDTs, when comparing our results with research presented in [203]. This contributes to interesting basic measurements that confirms and adds to what we have used.

In spite of our different assumptions and approach to the subject, it is unavoidable not be inspired and influenced by Vegas work. The fundamental difference between our research, which provides guidelines, and her empirical state selection method, is that the belief system in Vegas work is based on the notion that testers and developers are sufficient knowledgeable to actually choose TDTs. Our work has shown that this is a false assumption and if so few actually know how to apply techniques – it also means that they are not using the techniques correctly – and thus are not in good positions to make decisions of which TDT to use. This is supported by the statement by Vegas that "it is not techniques who find faults – it is people". This is a statement which could be interpreted as implicitly meaning that the people are not explicitly following techniques. In that aspect, much of the selection support fails. In addition, this also makes it clear that what it means to apply a technique differs between individuals, and consequently there is a variation dependent on how the technique is applied for a specific system.

13.7.1 Vegas Test Technique Schema

In this section we will describe an abbreviated version of the 34 categories in Vegas Schema Instantiation [203] (Appendix F) she used to compare some selected TDTs. Since this is the closest recent academic related work we have found, we have attempted a more detailed scrutiny of each of the categories one by one, indicating if we concur or discard it – and why. Our goal was to reuse and keep

building the body of knowledge around software TDTs. The below descriptions assumes that you are familiar with her contributions.

1. The *tactical objective purpose* of any TDT is to create a test case that executes some aspect of the software, and shows that the particular selected path works (which can be shown by coverage) or to find defects (assuming the software has a defect). Since this is the same answer to all of the techniques, it holds value in separating the different techniques, other than stating the obvious objective purpose of testing. Therefore this category is dismissed since it contributes nothing.

2. The *defect type* is fuzzily described in Vegas schema. She categories defects as "control, processing and any". This bears little information – and aids nothing in the goal of "finding failures/faults" in a system. If a thorough list of faults (in code) and their propagation (result visible for a test case in a dynamic execution) would be available, a taxonomy categorization would be much more interesting. This is still not thoroughly developed. In chapter 6 we have discussed this in our study, and our conclusion is that this is not sufficient. Therefore this would be a sought after feature, but at the moment the defect type is uniquely defined, it is therefore not useful or contributes in our schema.

3. The *tactical objective effectiveness* category is interesting. But we dismiss the measurements suggested by Vegas. Effectiveness of the technique, i.e., does it add coverage and does it find failures? This is of course totally dependent on the failures existing in the system, and how the measurement of the technique was conducted – as we discussed thoroughly in Chapter 6 [], where we concluded that there is currently no way to scientifically define the number, types and propagation of "all types of faults" in a way that would result in a fair judgment, whereas the SIR [], Siemens [] and Space []-suites are all much too limited in both number, type and distribution to pose sufficient answer to that. Thus, all answers in this category must be considered to be "indications" for a particular system. We have in our studies showed that yes, they do find particular problems because there were faults in this particular category. Can you conclude that it is a better technique than others? If looking into subsumes hierarchy of TDTs, this might partially answer this question. We will let this category remain, but will instead not suggest the percentage as an absolute conclusion, but as a "judgment". Any test case will add

coverage (if you start with no test cases) thus all test cases are in one aspect "effective" to some extent. Secondly, this category is more interesting if it can "add" to the coverage assuming some initial coverage exists. And sorting our dependencies is currently not feasible. Our best attempt is therefore adding coverage based on techniques in the context of our guidelines.

4 and 5. *The tactical scope* attributes are elements and aspects. In general, this is defined as "setting the test in space and time, specifying which part of the software it affects and development phase to which it refers" (Vegas [203] p.97). Scope is what we define as phase or "level" of test (in a test process), since most TDTs are useful at all levels. The scope of the technique is in principle dependent on the software, the way the system is constructed, and the know-how of the technique and the comprehension of the user, as well as the tool support available at the particular point in time, level or phase and also what type of faults that propagated into a visible state for the test technique to capture. All test cases are penetrating both the interface and the underlying code, making this a rather hard to grasp concept with regards to TDTs. We initially assumed TDTs were only "useful" at one level, but our investigations have shown that most techniques can be used at most levels. The one technique most people take as an example is "code coverage" as a level usable for only unit test – but as we have shown in our study in Chapter 10 is that usage and know-how of code is an asset for a system tester. Again, if it is possible to target and make test cases based on coverage decisions in large complex systems is more a tool support question, than an attribute to the TDT. Therefore we are dismissing this category as important, since it displays more a historical view of competences and limits in tools than the specific techniques used.

6 and 7. The *Operational Agents* (tester, developers) *knowledge* (6) and *experience* (7) is hard to measure. In industry, a person can have tested (or coded) systems for 30 years, and yet be unfamiliar with the most basic TDTs and still knowing the system well enough to contribute to "show it works" or provide valuable test cases. Yet many call themselves "experts" in test without ever having attempted to test a complex system at all levels, but are often instead "experts" in prioritizing, planning, coding, constraints or interpreting requirements. In our study on systematic mistakes [70] also described in Chapter 7 and 8, we found that applying TDTs is not an easily

measured competence. Therefore we prefer talking about comprehension levels. We further assume that the "use" that Vegas describe in her schema is based on knowledge already possessed by the person. We mean this if you are about to make a graph, then you are familiar with how to make graphs in this representation – and thus turn that into test cases.

8-12 are the categories for *Tools* (operational) including *Identifier, Automation, Cost, Environment, and Support.* We can easily say that tools support for TDTs is a large field that requires its own specific research. We regard this as a separate question and an ever-changing area. We dismiss Vegas approach of combining the technique with tools, since the list is both incomplete – and different tools interpret (instrument the measurements) differently for the same technique. Thus in our Coverage tool in Chapter 9, we could find that coverage tools that measure the same phenomena on the same code and software are giving different results. Thus, the discussion of certification of tools comes into play; who "owns the right" of defining the technique? It is not that interesting what tool you use for e.g. test execution (regardless of what technique is used, which are questions posed by Vegas) [203]. The tools will remain more bound to the system and the tool used. Evaluating the result (visible or log) is also arbitrary to technique, but different tools gets different results, based on e.g. instrumentation. Furthermore, a tool might be dependent on a specific format or type, which, if transformed to the tool type, might skew the results. We are instead of looking in to availability of tools guiding the test case creation – more interested in if the technique in itself can be divided into steps – and that these steps can be automated (or "rule-based"), and what needs to be done to actually automate the test case created. Thus, as we define – we describe this category differently. The technique in itself is the most interesting part in our Taxonomy.

13. The operational technique comprehensibility we have defined by adding the aspect of "applicability", i.e., can people apply this technique? And this is the underlying comprehensibility of the technique. In our studies, most subjects (testers and developers) have had problems in applying some TDTs, but we are still unsure of the reason for this. We support the category, and divide it further: Comprehensibility: (high, low) and add level of comprehension (A-D) as defined in Figure 13.3:

14. The cost of application of the test technique is so strongly related to the comprehension, but we support Vegas category, even if we feel this is totally system dependent. Thus we aim to comment this further in Chapter 14.

15. The "operational technique Inputs" is the category where the technique is "located" (see more description in Vegas) – since in modern system – all information can be "on-line" in different systems, it is not an easy to use category – where one can claim "specification" for everything. More interesting would be our initial division if the technique more focuses on data (input data) or on path (structure).

16. The category test data cost is we feel a bit bewildered about how this is defined. We concur with the senior researcher that this is non-measureable and can be discussed – and thus dismiss the category. It would be more interesting to claim "how big is the domain of inputs" for this technique – but that is also unanswerable – since this is defined by the frequency and occurrences in a particular system.

17. Dependability is by Vegas defined more as an additive contribution of the technique. This is rather fuzzily defined and investigated. We dismiss this category, and instead are suggesting the order (of adding) test cases in our guidelines.

18. The repeatability of the test technique is weirdly phrased. Every resulting test case should be repeatable, but here the actual technique is in focus. Once a test case has been created it can be repeated for every test execution (assuming something changes in the system). But for a specific input and/or path – it cannot by default be repeated on the same input and/or path if not anything has changed – that would be a plain "duplicated" test case. In that respect, it is an enigma what is meant by this category and what it contributes to. In her schema Vegas claims that "boundary values" is a technique that cannot be repeated. But to our knowledge a system can have boundaries as defined in the code, but also boundaries that propagate to the user interface, which definitely indicates that "all" boundaries are not necessary the same thing from a usage and code point of view. This dispute is easily settled with a very well specified coverage tool – that can show that two test cases in fact are the same, or have overlaps. But that depends on how one defines a boundary. Harry Sneed, claimed at the Vienna Test Managers meeting 19th June 2009, that there are in fact more than 10 boundaries (did he say 13 exactly?!) for

each “boundary” if one should take all aspects of software into account (byte-order, ASCII, Unicode…) and other different representations. This claim is rather unusual interpretation of what many consider a “well-familiar” technique, but nevertheless it depends on the context of test – and can very well be defined as correct.

19. The Technique Sources of Information is ok, though, we are recognizing that the techniques have evolved, and have multiple and more recent sources.

20. The Technique Adequacy criterion, but has to do if it is possible to define in an objectively measurable way. And – since we have shown (chapter, x z) that this is not feasible without certified tools, it is in our scheme pointless for other than coverage measurements.

21-22. Test Cases: Completeness and Precision seems to not been used by Vegas – and is for most systems infeasible with current measurement techniques in conjunction with the size of most industrial software.

23. Test Cases: Number of generated cases is inherent to the system under test, and the design chosen. It is interesting that Vegas provide a “theoretical limit” and a formula that could be based on measurements collected for the systems under test should be feasible. However, we have not pursued this line. It makes then the “practical limit” suggested be restricted to the system under test, and not a generic result, though heuristics is always used within industrial organizations based on some experience and collected measurements.

24-28. The next four categories for software testing techniques in Vegas schema, is software under test, thus the object. Applying a test technique for a specific object (system or software under test) is always done in the context of the object. Unfortunately, there is no generic classification of systems that can be used for this manner in relation to TDTs, and frankly the variety in the implementation of the test case using a technique is always so specific, that no generic solutions are to be found. It is assumed, that testing for a specific construct on code-level is of course related to that code-level. In principle – we regard any information that might be specific to system under test more a reason to exclude this as a generic technique. Thus, we assume that the technique is only valuable in a context it works, where an example could be “branch or decision coverage” that is of

course not measurable for programming languages lacking this concept. Therefore we are not dwelling on the object, and can comment if this brings any interesting information on the table.

29-34. The final "historical" and personal views of a TDTs bears little meaning for us, since we do again believe that most testers and developers using these techniques have limited know-how of them in practice. Thus we disregard most subjective evaluation as contributing to the technique to be discussed more in Chapter 14.

13.7.2 Beizer's Testing Techniques

Boris Beizer's has long been the master and leader on test design, summarized in his book Software Testing Techniques [19] from 1983 and updated in the second edition 1990. He claims have shifted from only viewing testing as something for programmers to now including the new breed of "testers", and also adding the fast growth of tool usage and automation. More interesting is the groups he uses. While sticking to "Flow-graphs and Paths" denoting use of code aspects, which we often call structural testing, measurable in the coverage. Structural testing is now expanded to all types of structures. Beizer separates Data-flow testing, which we also combine into structural test as well. Both of these groups would be "sub-groups" Functional structural testing are well specified techniques for given code. Different in nature is Beizer's Transaction flow testing group, which means testing databases. This includes e.g. inspections, reviews and walk-through's and together is currently not viewed as a specific TDT in our world. Testing databases is instead a specific type of application or architecture that have typical local functional goals to fulfill, in the same manner as testing a GUI, testing an operating system, as compilers, knowledge based systems etc. At its best, we would include it in "domain" testing. Domain testing, at least in our point of view, often denotes a specific type of industry, e.g. telecom, automotive, e-commerce etc. But again, domain could be anything we define the "domain" to be. Beizer defines domain testing as being based on the specification – and if the test is based on a specification – it is a functional test technique. Further, if domain test is based on the implementation it is a structural technique. This argumentation basically means that functional test is based on a written or model

description of the system, in contrast with the "real-life" implementation, which is structural. This view is not holding today. He later re-defined this view partially in his book "black-box testing" and thus contrasts that to "white-box testing". This terminology has the same limitations as we discussed earlier.

More interesting is the group called Syntax testing – that we kept similar and expanded, and as we will see, also is kept similar to Ammann and Offutt, who expanded into mutation testing. Due to this, we also expanded it into high-order (semantic) mutations, going beyond syntax. The description of syntax based testing according to Beizer is often taken over by the compiler, just as the string-based notation.

Beizer's group "Logic-Based Testing" mostly consists of decision tables and "State-transition" group is in our world collapsed into one group. Our viewpoint is that these are different representations (tables versus graphs), but they can easily be transformed into one and other, meaning the same thing. In logic-based testing, the decision tables are really tables of "rules" often used that defines specific fulfillment of different. One can discuss specific "conditions" as Beizer does in the form of predicate testing and this technique is defined in a rule-based manner. In principle this is basic condition [] in the coverage model, but can of course be used for defining a series conditions. Mostly – these are today implemented in simple coverage tools.

In conclusion, Beizer's test techniques are very specific, but often detailed on the code level, and thus many of these "techniques" have been hidden today into tools.

13.7.3 Ammann and Offutt's Hierarchy and Naming

Ammann and Offutt's book "Introduction to Software Testing" [4] is attempting to renew the testing scene. In fact, is the currently most educated and accurate book in many respects. It still repeats most of the types of techniques defined by Beizer, in a more condensed way, and updates them in the light of different coverage techniques, adding new use. The taxonomy seems to have many similarities, but is a modern update of Beizer's classic testing techniques book.

This means that the naming is preserved almost throughout. The Beizer "flow-graphs and paths" has now become graph testing (that we call "path testing" and similarly the Beizer logic-based testing is now transformed into logic (condition based) coverage. The interesting view proposed by Ammann and Offutt is that graph-testing includes state transition techniques (FSM's). We presume this is based on the fact that often these techniques are viewed in a graphical mode, or that the logical flow, path or graph dominates over input. Going from this view points, strengthen our taxonomy view that state-based techniques are indeed a subsection of the specified group (positive testing), that in its turn could be divided into input dominating or path dominating (as explained in Table 13.3) or both.

Domain testing as Beizer calls it, is re-written by Ammann & Offutt as the input space partitioning, that we plainly call input testing. In context, the input space partitioning could be viewed as the technique equivalence partitioning. It does only specifically state combinations of techniques in combination with input selection (like e.g. constraints), which could easily be expanded into a lot of other combinations.

The syntax based testing of Ammann and Offutt is the same as mutation testing, which is the accurate and timelier description of this group of techniques, compared to Beizer that we adopted and expanded. In conclusion, Ammann and Offutt describe their techniques in the light of coverage measurements, an approach with many benefits.

13.7.4 Murnane's BNF Approach

Murnane's [157][158][159] BNF "Atomic Rule" approaches test design from a teaching perspective. Murnane attempts to define the BNF grammar rules for each of the classical input techniques, and successfully does so for some techniques, particularly equivalence partitioning. The basic idea is that since the techniques are well defined, the result could be defined by clear rules that might in the future lend itself to automation of the test case design. This is an important contribution to the testing field.

13.8 Discussions on Test Design

There have been earlier studies that claim similar results when one applies a technique for a particular system compared to ad hoc testing on the same system. This must be compared to our studies, where we can show that the result seems to be much more related to the knowledge and comprehension of the testers (of both the TDTs and the system). The same test technique can span between 2 minutes to several hours to create. Is this a result of the system, the comprehension or the experience? Currently this question needs much more dissection on a scientific level to sufficiently provide a straight forward answer. Our view is that "it depends" and the list of what it depends on is just hinted in this quasi-experimental studies. Comprehension seems to be a key ingredient. Therefore our goal and aim slightly changes, as we attempt to justify here below.

13.8.1 Why Guidelines are needed for software test design and changes of expectations

In our research we have investigated the use and understanding of different TDTs. In addition, we have supportive evidence for a strong bias towards usage of what is called "positive" or specified testing in our studies in Chapter 7 and 8 compared to earlier results [194]. It turns out that most testing performed in the software industry is positive testing, i.e., showing that something works. For software systems that exposes vulnerabilities, handle finances or other sensitive information, this is clearly not sufficient testing, and especially if exposed on the Internet or with open interfaces. At the same time, the assumption that the user should behave and use the system "perfectly" becomes more and more invalid. Earlier users accepted in depth training for years as a must to use some system, but since software systems are everywhere we also adapt to different expectations. Systems should install automatically and fast, and the commands to the system should be input restrictive and intuitive, making usage of the system easy, with a built in way to avoid faulty actions and inputs. This also changes how we test and how we view the user. Robustness is the quality that it should not be possible to destroy anything by accident, and prevent any malicious behavior.

A common view is that if you apply a new test technique then you will find new failures. DeMillo [8] claims that any new technique or test approach will have peak failure finding abilities for a while. This motivates teaching of several techniques and invention of several techniques. James Whittaker [5] claims that even if you do not apply any technique at all, you will sooner or later stumble across a software failure by mere execution of the system. It will not be effective or efficient, but this is one of the reasons many keep a very low regard of testers – how hard could it be to just execute the system? You will find something eventually. On the other hand, some industrial systems have good reporting and information gathering about the systems. In telecom networks the problem is almost vice versa, sometimes the tester is flooded with information, and the problem is to identify the reasons a lost call really was lost. Is it radio-shadow because the phone was out of range, or is it the software that dropped it? Or was it the user that turned it off? It is not obviously easy to differentiate what is a "real" failure, and what is an induced "legitimate" failure. When investigating software faults and its propagation into a visible failure – and what technique that would have exposed such a failure, it seems complex, when at the same time some faults never come into action for different reasons.

13.8.2 The Impact of System Knowledge on the TDT

How important the know-how of the system is, and when (and how) do we learn the system? It should be obvious that we learn the system by executing it (or similar or earlier versions of the software system), instead to learn it from "reading specifications", which in many ways can be deemed inferior. System know-how and particular how the system acts whilst using it, must be internalized knowledge of a tester, together with a deep understanding of how the software behaves, and how software failures can show themselves [211]. Regardless of if the test case is written down or not, it will only be as good as the persons knowledge is. The advantage with writing test cases down, is that each test case can be reviewed, and improved by others, and thus provides a specific learning for the person writing it. It is hard to follow a random execution on software that is neither repeatable nor

detailed enough. Of course, tools can be used to capture each action, and then store that for later scrutiny, which makes test cases also possible to keep for regression tests. Having looked at captured user actions, they are not easily readable, and rather difficult and costly to investigate and learn from. Another aspect of exploratory execution is that many systems lack a full-fledge user interface, and most testing is by making fake interfaces between "computer to computer" software. For instance, in one of our example systems [OSS] only a fraction of the system capabilities were understood and reachable through the interface, and not until the code was inspected the system showed its capacity and faults were revealed We conclude that one should not exclude on approach for the other, and probably are both approaches needed for different reasons, and depending on the system, competence and situation, but that exploratory is just using ordinary TDTs, and is not a particular technique in itself, but a new context in which to utilize TDTs. We can conclude that learning the system might be a part the test approach, but we have excluded it partly, when selecting the test case and investigating TDTs. Exploratory testing does instead questions the need of writing down test cases manually [60].

Instead we have in this thesis made a great effort to define what a tester does wrong when writing test cases [], to better pin-point and understand the improvements needed. This also shows how important documentation is – and writing test cases down. We want to note that it is of course possible to capture the exploratory test and repeat it, but systematic walk-through of the system will probably be lost without relying on some structure, specifically, testing only through the user interface, when the system is integrated and available. This limits early feedback on quality, and does not rely on capturing intermediate faults.

Chapter 14. Guidelines for Industry on TDTs

These guidelines are a proposal for how the Industry should use Test Design Techniques (TDTs). It is based on the research presented in this thesis, together with our experiences of testing complex software intensive systems. Our goal is to assist in finding out which techniques are good for what, and when they should be applied, as well as to show which TDT is most effective and efficient to use. This guideline should be useful in a majority of software systems for commercial and industrial purposes, where the quality of the software matters. It is proposed as a general guideline, as we assume that most industries are struggling with similar issues.

Our main assumption is that these guidelines are not limited to any particular system or any particular level of testing, but there might be reasons why a particular advice will not be applicable in a specific context (e.g. it is not meaningful to improve decision/branch coverage in a system with only straight sequential code or truly parallel code that does not include branches). As we have described earlier, we boldly claim that there is no limits of using different techniques at any level, but there is often a limit in the comprehension of using these technique. Many would claim that for a large complex system, e.g. instrumenting the source-code (or byte-code or intermediate code) with coverage information is infeasible, and thus un-measurable. We claim this is not necessarily true, and for a testing purpose a near enough code coverage measurement could always be achieved. Others might feel a limitation in tools, know-how and other aspects – but these problems can be surmounted. The true distinguishing difference between success and failure of testing is how comprehensive quality awareness the organization has and ability make improvements. Finally, we also assume that the goal of people reading this thesis is a genuine interest in achieving improved quality. We provide no arguments for testing less – which just results in costly maintenance and loss in confidence of the software. The main goal is, and must be to increase the amount of testing in a cost-efficient improvement

manner as much as possible. This argument is based on the knowledge of the cost of poor quality and maintenance. Another way to view the same argument is that effective and efficient TDTs will get you to an acceptable quality faster, thus improving the time to market.

14.1 Test Design

The main target with test design is to define different goals for the quality. Since the core of testing is to create a measurement of quality, i.e., to create test cases that include executing the system and get a measurable verdict that contributes by satisfactory showing that the execution is according to the intentions and goals set. This means in practice that test design is the main phase where to define how and where to create new test cases or to be able to select a subset of test cases to be used in the testing. Furthermore, test design can be repeated over and over to fulfill the goals stated in the test plan. To create test cases should be done as effectively and efficiently as possible, assuming they are applicable in the context of the system.

The main goal of testing is to give enough time to measure and demonstrate that the system fulfills given quality goals, i.e., fulfilling all test goals defined (measuring the quality) of as many aspects defined and described in the system as possible (within given time and resource limits). This is essentially is the same as to

- Get as high coverage of the system or object under test as possible (demonstrating that all targeted (and other) aspects works and thus finding as many problems as possible)
- Focus on using as little effort (cost) as possible, thus adopting a high degree of automation in as many steps as possible; this also improves repeatability and security.

Since there is a trend to define “risks” of the system, and create “priority” of importance of different aspects of the system, one must know that these exercises are more a way to engage the organization in the task of testing, something which in many companies is often not sufficiently evaluated as an asset. One should carefully consider the time spend compared to the aspect that the time could often better be used creating more test cases, based on the notion that we have

confirmed with evidence that people seems to create most test cases based on obvious and specified aspects of the system. It is often much more difficult to focus on systematically testing the entire system, and to make sure that no aspect of the system goes untested, without a proper analysis of the consequences. Depending on the goal of test, our guideline is made to support an increased order of improved testing, assuming that the know-how of these aspects are understood (which in itself seems to be the largest problem). One should make sure essential basis of testing is well understood by the people who actually test. Our contribution is focused mainly on the specific TDTs that are used to create test cases with, and we have almost entirely avoided to go into "project", "management", "process" and any other organizational aspect of test – which in almost all aspects resembles its counterpart for developments (except that the skill set and artifacts differs slightly).

The result of this guideline is a proposed order in which TDTs should be applied, where each refining criteria will yield new and additional test cases. The guideline also includes a first discussion of what TYPE of system the approach is valid for, thus considering the specifics of some types of systems.

14.1.1 Scientific Framework for the Guideline

One of the major validity threats of our guideline is that even though the current body of knowledge is large, it is hard to show that (anything) is valid for all software systems, all organizations, and across all cultures. Despite this risk of limited validity we have decided to present our guideline, because we both have a strong belief in its validity, and since we are convinced that even in case of limited validity, it will help improve industrial testing practice simply by providing a structured approach to TDT selection. This guideline is based on three sources:

A. Our research papers (will be referred to by Chapter)

B. The written body of knowledge that compromises, standards, books, research papers and other documented evidence. This will be referred to with proper reference.

C. 30 years of experience in the field of software testing in a series of industries, and in multiple types of projects and software systems (e.g.

telecom, administrative, transport industry, banking, open source, embedded systems). These will not be referenced, and if a claim is given without a proper reference, it most likely belongs to this category.

The different claims in the guidelines are derived from the above three area, and the guideline is writing primarily with an industrial audience in mind. These are an initial set of guidelines that need further exploration, case studying, and feedback, but to our knowledge the advice should work in almost all cases. The main idea is based on a concept of successive improvement, i.e., that you compare your current status with what is suggested, and then improves successively as the guidelines suggest.

14.1.2 Test Design Target Area Overview

Test Design targets several areas – and describes what and defines how the system should be tested. Therefore we address different stages of the test process, where how you design your test impacts the quality of the result. In many project, selecting tools, process and requesting appropriate funds and resources, to do adequate testing, is a phase called Test Requirements, and is often prior to Test Design (see Chapter 2.4) It is assumed that the system – and proposed changes and requirements have been properly analyzed and understood by the testers and developers. This must be the case even in a very parallel development and test environment. In fact, choosing the wrong tool, lacking to educate the people on the tools, or not providing appropriate processes can havoc any test organization. The scoping of the effort, organization, and sufficient resources must have been estimated – this is known as test planning [110][163]. Depending on the complexity and quality goals of the system, a sufficient “divide and concur” approach must be taken. There is a lot of know-how involved with the test planning, test analysis and test requirement phase, but we will not discuss this further in this thesis. Instead we target some specific notions that must be considered basic knowledge, e.g. “What is a test case? What do we mean with using a TDT?

14.1.3 What is a Test Case?

We assume a basic knowledge of what a test case should contain (as defined by the test specifications and test procedures in IEEE 829 [110]. To this end we provide a test case template in Appendix 1, and in Chapter 11 we identify some of the most common mistakes made when defining test cases.

Yet, we ask us the fundamental questions: How big should a test case be? Or in other words: How many steps and actions are enough to make it a test case? Current "rule" is that anything from a signal, acknowledge or just a "key stroke" or "mouse click" can be considered a test case. In industry, test cases can be as large as a 400 page detailed description document or a test program (e.g. test code) of several thousand instructions. There is no coherent "rule" of what is a correct or reasonable size of a test case. Rather than discussing size *per se*, one must consider aspects such as:

- What the TDT implies is the correct number of "steps" to fulfill the intention of the technique
- What the goal (result) of the execution of the test case is meant to be
- What is a convenient size to manage; should not be too small to cause a test case explosion, and not too large to contain overlapping steps between different test cases
- What facilitates automation of the test case – e.g., to treat the data as variables, and to let the test case be the unique steps to follow.

These goals and aims can sometimes be conflicting. One example of such conflicting goals is the boundary value test design technique versus the goal of avoiding overlapping test cases. The minimum number of test cases for both of these techniques requires executing the same number of steps, but with at least two (but for automation purposes rather three) different input data. From an automation perspective it means that one step by step description can be executed twice (or many times) with different input data. Each of these executions could lead to totally different expected results. In reality this would mean two (many) different test cases. From a TDT point of view, this is only "a single test case" (but with intermediate check-points), since if not run together it can be hard to establish that the goal of the TDT was fulfilled. This touches on the definition of what

a test case is. Do we make a test case an “implementation” with a explicit data and its result (this is the most common interpretation) – or do we mean that a test case is a series of steps, where the input can vary, which is the automation view of test execution. Further – the test case is not only about input data, even internal order of test steps can impact the result. Basically, permutations of step-by-step description (i.e. order) could cause one entirely new set of results in software. Would the new order of steps then be considered a “new” test case? And is the answer no to the former question only when the expected result is the same? Is it the actual resulting state (expected result) that determines if we separate them? From an automation view they would be overlapping (if they contain the same steps). But does that then necessarily include preservation of order of steps? The lack of being precise is evident, and matters greatly for our success in producing efficient and effective test cases.

This gives a first overview of the problems related to describing test cases, and many of these questions are a start of uncharted territories in research, where currently the most common answer from a practitioner to all of these questions will always be “it depends”. Where the factors of what this depends on, are neither written down, nor significantly researched and concluded, and have not even full answers for specific systems.

14.1.4 Test Specifications

First and foremost is to specify test cases such that they can be automated. This means that the test specification should contain a series of important information to ease the test case creation, but furthermore, if defined correctly and linked with traceability then we get automatic debugging support, including fault localization. In particular the first part of a great test design is defining the test specification – the source from which the implemented test case (test procedure) is created. The test specification is at a “high level” and in complex system it may be described by several “multi-level” documents. The test specification should describe “all tests possible” to fully verify (execute and determine) the software under test.

The test specification should specifically:

1. Contain administrative data for configuration management. In particular the data defined in the IEEE Standard 829 Test Documentation (1998) [110] as presented in Chapter 11. All aspects are assumed using normal document handling standards – thus contain creation, review, update information e.g. when, by whom, in what project, version control, storage area etc.
2. Define templates, model and tools that provide a uniform way to present the test specification, and are managed such that changes can be traced.
3. Contain clear identification and traceable information to requirements, system architecture/design etc. An update should add not only traceability to the documented structure (requirements specifications, design specifications etc), but also direct traceability to the software structure (this is especially important if any sort of legacy system is used, modified or expanded upon).
4. Identify important aspects in the specification, such as priority, candidacy for regression test, and in what order the test suites are to be executed. This could be practical information like identifying the system/development test group, where e.g. performance test is often kept as separate group.
5. Contain one or probably a series of descriptions of executable paths or actions in the software, for which a particular function or aspect is pointed out. Options of alternatives, repetitions, etc. should especially be pointed out in a case-by-case manner. Often this is referred to as "use-cases" or "user-scenarios" – but on many lower level of software there is really not much of a "user" visible. Defined as a series of "execution paths". This step is particular suitable for modeling, as long as the tool in itself does not become a hindrance. A more detailed description of this is:
    a. Specification of expected/default/normal behavior (functional and non-functional) and limitations of input.
    b. Specification of abnormal/invalid behavior (functional and non-functional) and for input that when it is the case.
    c. Specification of execution paths (functional), i.e. valid and invalid paths.
    d. Suggested permutations or parallel areas – where execution can be done in parallel or in arbitrary order.
6. Contain specification of time and resource dependencies (when applicable).

7. Contain specification of environment and context.
8. Contain specification of applicable standards (if any) in addition to IEEE Std 829 Test Documentation, and for system test purposes ISO/IEC Std 25000 (former 9126).
9. For each and every input – define a thorough input analysis.
   a. The input analysis should include valid and invalid input, as well as boundaries if applicable.
   b. In case of several sub-groups of valid input, each sub-group could cause different actions for each individual group of input data.
   c. The input analysis should always include the entire ASCII-table of characters, and also define maximum the bit/byte/length/nr of characters (or minimum) allowed for the input.
   d. Actions for invalid data should be defined – and could also be handled in subgroups. Depending on type, this includes if more characters/bits than allowed is used.

Test Specification could be (and maybe should be for complex systems) divided into a multi-level document as previously stated. This means practically, that the test specification for system requirements could be grouped depending on the main characteristic issue it caters for. A common grouping is based in the ISO/IEC 25000 (former ISO/IEC 9126) [117] where the areas are e.g. Functionality, Robustness, Installation/backup and administrative, Maintainability, Usability, etc.

Furthermore, all these aspects could also be grouped and divided based on the software components' logical or conceptual structure. The advantage of grouping is that things belonging together are kept together – and one can talk about "competence" areas in a complex software system. This sort of organization makes changes in the software system, and changed functionality or conceptual changes harder to create in the structure; which seems to make people feel ownership and responsibility of their work.

Division on complexity and comprehensibility level is inevitable. The more complex the software is the more important is it to divide or "scope" into "levels". The comprehension levels are often mirroring the know-how described in Figure 13.4 above. This means that for some systems there is a code-level, one or many integration levels, and one "sub-system" level (that from within is claimed to be a

system level) but is then integrated into a larger system or a system of systems, depending how to view it.

Therefore, it is important in this context to talk about “levels”, which are clearly a consequence of the complexity and the structuring of the systems. Thus, how to divide and how much information that is redundant, complemented and needed, is truly an organizational question that is best answered in a case-to-case basis for a system.

14.1.5 The Test Case Documentation and Automation

Two parts of Test design is Test preparation by specification, and test case construction. We separate the issue of applying test design into two parts: Documentation and Automation, where we here discuss the documentation and design context of the test case, thus, the information that a test case must contain. The level of documentation differs if the test case is to be information to a human or a program to control a computer.

Many believe that automation is a time-consuming step. Instead one should make clear that many manual test cases are often an ambiguous implementation of a test case, and thus contains a series of flaws (see Chapter 11). Thus, an automatic test case is unambiguous – and can be repeated by many times, fast. Since manual test cases are ambiguous, the interpretation becomes a know-how either kept by the person creating the test case (which means it will be different execution if one other person executed the same test cases). Keeping up the hidden information or know-how about how to interpret test cases might be costly, thus the quality of the test case becomes person dependent. Secondly, most people get tired and bored with repetitive tasks, and finally, it is also common/possible to make mistakes and miss actions that can turn out very costly at a later stage of the testing, either unexpected behavior happens, or it might not be clear if the execution was performed adequately. In organizations with a low degree of test automation, a lot of time is wasted on repeating the tests again. If manual test case documentation is maintained, the problems defined in Chapter 11 and above, must be carefully considered, and a review of the test cases must take place, making sure that flaws do not remain in the test case documentation, thus an unambiguous way to

define the test cases must be provided. How to structure and execute a series of test suites in a particular system is outside the scope of this thesis.

14.1.6 List of Advice to Improve Test Design

1. Test Cases should contain all aspects defined in the IEEE Std 829 Test Documentation (1998)[110] (e.g. Test Specification, Test procedure, Test instructions, Test Data)
2. In addition, administrative information should be added. One example is the time to execute the test case (providing a basis for optimizing the testing order based on execution time). If all test cases have the same (or insignificantly different) execution time, such optimization is not necessary. Another addition might be "assumptions" or "pre-requisites". One often assume a specific circumstance when writing a test case in a complex environment, but if e.g. a simulator is used, the timings might be off completely, and the test case not really fulfilling its purpose (or getting a skewed result). Priority or grouping/type of test case is often beneficial. Some of these aspects are visible in Appendix 1.
3. Which TDTs to use when creating test cases is suggested in different lists in this chapter and should be made a conscious act when creating test cases. We have added information based on efficiency, effectiveness and applicability (comprehension) to add to the body of knowledge. See further in Chapter 13 (as referenced).
4. Test cases should preferable be written immediately in code (test code), and organized in different abstraction-levels. This means they should be written in a manner of code or pseudo-code as a clear instruction (e.g. test code). This is purely based on efficiency studies (diminishing steps) in industry, but has not to our knowledge been confirmed in research.
5. Connecting a particular test case not only directly to a test specification but also directly to the software structure makes it easy to locate and identify what you have executed. For integration and some types of functional executions, such as use cases that often spans over a series of software parts or

components, it is important to identify all involved software parts for each step. This can contribute to an automatic and fast debugging if problems are found. Automatic traceability to the software structure/code item should be defined in the test case. See Chapter 12 for further advice.

6. The test case should contain (as defined in the standard) a starting position (which could be one or more executed test cases), a step-by-step action, a selection of input, and a verdict (or a step-wise instruction for how to conclude that the test case passed or failed). This could be including automatic analysis of logs, measurements etc. These actions are confirmed by the test standard [110] and our study of use in Chapter 11.
7. It should be defined what needs to be "removed" or cleaned up from the system to return to the position before the test case. Many software systems could also benefit from "not" cleaning up, e.g. to investigate suspicions of build-up in heaps, stacks, queues and also memory problems, in addition to fragmentation problems, filling up databases and lists etc. Consequently the test should be marked for use in a regression suite or not. We have discussed aspects of "clean-up" after a test case in Chapter 11. Removing or keeping data (for possible build up) seems to be a system dependent feature, to be further studied in research.
8. If total suite(s) of test automation execution is built up, there are other factors that is important for the testware aspects of the software, such as tools used, test architecture, use of libraries, minimizing test case redundancy, education in the tool, and a similar structure of creating test cases. In addition, test management (configuration management, as well as data handling, propagation and results aggregation and presentation) becomes more and more important as the test suite grows.
9. One should not ignore the need and possibilities of using TDTs that automatically generate "half" test cases (just the actual execution), but then does not handle the "test oracle" – thus how to determine if the system/software has passed or failed. This must been taken into account in the test case. A test case must have a verdict to be considered a test.
10. Make the automation open for organization of the test case in the following manner: A test case is a series of execution steps, with

a set of inputs and its expected output and to vary the following parameters:

- Variation of execution order
- Variation of input (with its associated expected output)

This action will improve the quality, variation and challenge the system.

Following the above list will have impact on all aspects of test design that will improve the quality of the test and contribute to an efficient and accurate test preparation.

14.1.7 Order to Apply TDTs in

This section contains the actual description of the order in which we recommend TDTs to be applied to obtain the most efficient and effective test suite. We will additionally attempt to justify our synthesis.

Since people tend to test "the obvious" [194], a clear structural testing must be performed. The tester must be aware of each part of the software (at all levels). Attempts on minimizing the test selection by using "usage profiles" (or plain frequency measurements) can add information on customers' use of functions in the system, but new user groups bring new behavior and new challenges to a system. Also, one can assume that these "most common" areas of the software will be tested anyhow, and focus on other parts instead. Basically, any information that could impact the test case execution can be used. Even if e.g. "back-up" is a routine executed rarely, it might have a higher priority for other reasons, so frequency alone can never be the sole factor of priority. Often aspects of visibility, severity and business impact factors are equally important. Focus should be spent on the goal of the test, and given the order above, the level of where the system is and what the user expects. The above suggested test solution should be tailored to the system under test.

This list below complements the list defined on Test Design in Chapter 14.1.6. This added detail of TDTs will better describe our proposed test improvements, assuming a full comprehension of the techniques is available.

We will present the order in which to create and derive test cases or a test case suite. This is the order of quality improvement starting from 1 and adding successive refinements. Our recommendations targets functional test and is intended to be applicable at all levels of test.

Preparation:

1. Make a complete input domain analysis. This analysis means that a minimal test case execution should be attempting equivalence partitioning TDT. The input analysis can be defined as early as in the requirement or any specification (e.g. test specification) moment. Even if the very first step is to only execute the test case once (using one input) form one class as a first priority (see step 2), this technique will provide a basis for the next step of improvements. The equivalence partitioning is therefore our most prioritized and first technique to use, since all software needs its input domain analyzed. Equivalence partitioning will define allowed valid positive tests, and make negative tests (faults outside the expected behavior) less likely, since it transforms invalid to specified behavior. The best is to use this technique together with a random selection within each partition/class – through the use of a variable as input (discussed later). The TDT is generically applicable to all systems.

Execution order:

2. The first execution is showing the obvious works, thus use the default, normal value, e.g. the positive test, which implicitly is one value for the most common valid class of input (utilizing the input analysis results of classes). Selecting test cases can be obtained by using specifications, derived use-cases, or by modeling of the system. Also, using existing requirements, or any form of documentation can provide this basic simple assumed behavior, which is essentially what is provided (and expected) by any execution. This also implicitly defines the most important execution paths of the systems, depending on how many steps are in (size of) the test case. Make sure the test cases are not too big and not overlapping too much. A too big test case means that

many steps are similar (identical) in different test cases, and results in same execution paths.

3. If applicable – targeting a Boundary Value as early as possible (meaning targeting input borders) for numerical ordered sets where a selection of values (minimal 3 values for each boundary) is advisable, since such tests are shown to find important problems.
4. It is easy to miss that one input value for each input class should be selected in equivalence partitioning. Make sure to complement the valid (specified) selections based on the input analysis. Note that this also means, that at least one class/partition should be selected from the different classes that enforces specified fault-handling to be invoked, e.g. too large input or wrong character. It is here still important to avoid test case redundancies, thus 1-4 are essentially performed in one test case template with a series of values set to a variable as the input (and can be improved by adding a random selection from each input class to get the best execution value in an automated set-up).
5. Coverage shall always be measured for all test executions (at all levels of test). At least coverage of 100% of the feasible statements should be aimed for, and the additional effort needed to get branch/decision coverage is small – since it basically amounts for many systems to explore the accuracy of the equivalence partitioning as defined. This often amounts to a problem defining combinatory test vectors as input to fully explore the system, since many dynamic binding and late dependencies are hard to foresee. Nestled calls also bring problem for many coverage tools. Coverage can be a goal in itself for unit test on very small code units, but is best as a technique used complementary to other TDTs. Following our suggested order ensures this. Note that there are a huge variety of coverage approaches useful to successively improve the code. Examples are MC/DC coverage, LCSJ and some of the data-flow techniques can provide much insight in an environment requiring a very robust code (e.g. safety-critical code). Often it is sufficient to complement the above test cases by adding coverage as a guide of what have been missed.

6. Attempt to deliberately test outside the boundaries (see negative tests in Chapter 10). What is applicable is system dependent. These areas are both fault infested and important to target to create robust systems.
7. Permutation of test cases (non-dependent) can cause a varied execution pattern revealing faults. If automated, this might be an easy (low-cost) way to expose failures.
8. Since all combinations of execution paths (or input) are almost impossible to attain as a goal for any larger sized system, attempt to define and check as many important parts as possible. At this stage, a clearly defined model of the system (state-transition methods), with clear input handling will probably aid in obtaining a consistent and systematic generation of test cases. Targeting testing of the integration paths between different modules/units has proven to reveal faults.
9. Improve data-flow coverage by using a random selection from the classes/partitions defined. New techniques e.g. search-based testing approaches seems to aid in giving a fast and accurate set of important input combinations – which often becomes the second most important goal.

The TDTs are assuming repetition of test cases, and automation of the test cases is the most cost-efficient way to achieve repetition. Finally, the above list of functional test can be easily combined to measure the performance of the system, stressing the system, etc. Specific security tests must be added when the system has a vulnerable interface.

14.1.8 Motivation for this Order of TDTs

Equivalence partitioning is the best way to classify input (valid and invalid) that impacts the program flow. If size and type are explicitly added to this classification, you have a rather well defined set of values that targets both positive and negative test, and are by this making the best analysis of the program at this stage. The working method is that you start with (a) the valid, default, classes or partitions and then follow with (b) the invalid classes or partitions. The minimal first set is at least to test (a) one value out of each valid class (if there are more than one valid input class, choose the main,

normal or default value) as the initial test case. This should not be compromised on. Secondly, at least (b) one invalid input should be executed. Here a judgment must be used on the likelihood of the system executing the code being tested by the invalid test. There are often a series of classes for different types of negative input. Variation of size (outside the allowed size) is one aspect, as is looking into types etc. Therefore we added text in the list to make sure that the equivalence partitioning for negative test (variations) and classes are handled correctly. A good starting position for this technique is the ASCII-table, and as stated in the Test Specification, the analysis should have been done at this level. It is of course still possible that you assumed or defined these classes wrong, and that not all values within the class are treated equal by the program within one class. The best way to complement this is by adding a *random selection of values*. An advantage is that the random input selection combined to Equivalence partitioning is usable at all levels of test and all domains when input content matters. This can also be clearly defined in the specification.

A more specific technique, *Boundary Value Analysis,* should be used when applicable, e.g. for numbered ordered sets. This technique targets very clearly a common group of faults often on the "border" between two classes. Note that misunderstandings of this technique are very common (see taxonomy in Table 13.9). The technique is useful on all levels, but depending on the system and type of data handled, it might not be sufficiently clear how to apply this technique.

Code Coverage seems to give a good guideline for a "minimal" testing for procedural languages – or for languages with the emphasis on control flow. First all code must be executed at least once (100% code coverage). Feasible coverage is a better aim, since non-reachable code could exist. Non-reachable code is often referred to as "dead" code, but could be very important draft code to be used in the future, or security code (that is used to capture for random problems – but should "not be reached"). In the group of "non-feasible" code, is code that is very costly to reach automatically or by execution for some special reasons (that must be analyzed), but a way to ensure quality is that this code could instead be "reached" or covered by code-review. Code review is outside the scope of this thesis, and not considered a TDT, but is nevertheless an extremely useful method. Removing this "hard to reach" code can sometimes cause more damage to the

system, than leaving it in, and one must seriously challenge "must rules" in this area in research (especially for safety-critical systems).

The best aspect about coverage is that it is factual and measurable, but the down side is that even if at 100% it does NOT mean that the system is fully tested. The conclusion is that first the focus should be on equivalence testing, and secondly on coverage. If equivalence partitioning is used to its full, the Decision (branch) coverage, should be very high in its initial measurements. Therefore code coverage could be used both as a guideline for developers and testers at levels closer to code (functional, or at system level in few-level systems) to complementary test how well their test cases succeeded, as well as clearly point to what needs to be added. Limitations are usually the possibility to make instrumentation of code in large or time-dependent systems, at the system level. Data-flow coverage is currently not explored enough as a measurement, but shows promise. For a full-fledged measured coverage insight, look at [4][220]. The level should be building up the coverage based on the order of bottom up, using the subsumes hierarchy as defined in [30].

Note that *State based, state-transition modeling* (model based test), is a method that is best used with tool support. It is useful since it aids in understanding complex structures thought its often graphical displays. Tools should be used and judged on the ease they convey this understanding. How well do they communicate the system, and thus the test execution structure and insight in how items in the system are connected? For novices to the system, this is especially useful, and also in case of very complex structures. There is a big risk that the capturing of the model is wrong if done by novices. This is a strong research area, and it will soon be possible to generate models automatically from the code base. Use-cases (user scenarios/ execution paths) usually build up a model. UML provides one possible and useful basis to standardize the way to create these test cases. In fact, the plethora of languages, models and representations makes this a vivid technique – where the tool in itself often sets the limitations, in either modeling capabilities, visualization, loops, and abstraction levels used. Achieving 100% state-transition coverage should be relatively straightforward, thus covers the main paths. The greatest risk with model-based testing is that it by definition focus only on positive test and if not a proper specification is created of

what should not work, it will miss to encapsulate the negative tests, and this could easily create a false illusion of complete coverage.

The above main approaches will provide the positive testing, and targets the main combinations of flow and input in the system.

To focus on Robust and Negative testing, as well as combinatory testing, is secondary in priority from an industrial standpoint, but are important and investments in adding more tests in this area will save valuable cost in maintenance.

14.2 Applicability

Applicability is the ease of applying a technique on an arbitrary system and the ease of understanding the technique. The ordering of these techniques is the guide on know-how to be a basis of teaching these techniques. One example that shows that this is not arbitrary to any system is e.g. that 5 (branches/decisions) might not be present in all programming languages. Therefore we split the applicability into two ranking lists:

The ranking based on comprehensibility is the following:

1. Positive testing (any valid statement fulfilling execution)
2. State-Based testing techniques (use-cases, models)
3. Some aspects of Negative test (outside the boundary, e.g. overflow)
4. "Magic test set" (requires know-how of specifics!)
5. Coverage (specifically statement, branch/decision)
6. Boundary Values
7. Permutation between test cases (in practice difficult to shift steps)
8. Some of the negative TDTs
9. Equivalence Partitioning
10. Coverage (specifically basic condition, weak and strong mutation)
11. Fault Injection (as a theory easier to comprehend than Mutation)
12. Mutation Testing
13. Search-Based Testing

The list does not include combination techniques that can be both difficult and easy. This list are based our synthesis of the studies 1 to 9 in Chapter 3 to 11.

The ranking is based on ease of applying the TDT on any system is the following:

1. Positive testing (any valid statement fulfilling execution) which includes any state-based testing techniques
2. Coverage (specifically statement, but language dependent)
3. Equivalence Partitioning
4. Fault Injection
5. Some aspects of Negative test: Outside the boundary, over-sizing
6. Random, Mutation , Search-based
7. Some of the "Magic test set" (this is specifically easy, generally less possible)
8. Boundary Values (might be totally absent for many systems)
9. Permutation, especially within a test case (steps), that challenge the systems (but many system possesses strict execution order)
10. Negative testing (other approaches)

One can conclude that 1 to 5 in the above list is TDTs that can be done manually, and that 6 particularly set up for automation. The remaining TDTs (7-10) are difficult to apply (use) for all systems in a generic sense, but in the individual case could be very easy to apply.

14.3 Efficiency

The efficiency means how "fast" the technique is, i.e. is a measure of the rate of fault finding. A very interesting result is that we can group the total efficiency in relation to comprehension. This ranking is implicitly related to applicability, since fast also means fast to comprehend and fast to find a location to apply it in the system. Fast could also mean "fast to analyze and understand" when it comes to the system – which is often a hurdle in large industrial software systems. Of course any efficiency measurement can be measured in many different ways. Therefore this ranking looks somewhat different.

We could in our three studies (Chapter 8, 9,19) conclude that the efficiency is similar (no significant difference) between these manual input related techniques: *Positive, Negative, EP, BVA* or *Random Input*. The cost in time it takes is based on

- Time to understand the technique
- Time to locate the "next" candidate
- Time to do the analysis and select a value
- Time to set up the selection (for random) – which is then compensated for later in the time execution
- Time for execution
- Time for verdict

The cost for preparation of the test environment is not included in this timing – since preparation environment is not a factor of test, but a factor of the system.

Compared to this initial group of input focused techniques, the following techniques are more time consuming in its entire process, based on the cause explained in the parenthesis following:

- Coverage (analysis time of what is missed, use of tool)
- Fault injection (what faults to inject and where)
- Mutation (set up, build, tool)
- Random techniques (set up, verdicts, tool)
- Search-based techniques (comprehension, set up correctly, evaluate verdicts, tool)
- Permutation (intelligent challenges)
- State-based techniques (verdicts, tool)

With "tool" we mean that the entire process of procuring, evaluating and setting up a tool must be considered. Also efficiency is much dependent on the efficiency, comprehension, usability of the tool in question, and can vary between tools. Since this is often a onetime investment, one could debate if a fair "efficiency" comparison should be made after all is set up and the understanding (comprehension) for the selected technique is completed. Probably all of the above techniques would then be regarded as efficient, or strongly related to the selected implementation of the tool. Coverage is often viewed as a measurement tool, but is not all testing a measurement? A better approach is to initially attempt to apply the first group, and then just complement with the use of coverage tools as a complementary improvement technique. We could see a strong improvement in the efficiency with a manual use of coverage. The more familiar the tester/developer got with the tool, technique and software, the faster

the test cases were created. This is probably true for all techniques, but the difference were more noticeable when a more "slow" manual technique was used. Search-based techniques could for example be used to improve automatically on the coverage. It is better to compare efficiency within the group. Yet, the problem of deploying search-based techniques for real system is more related to the fit of the search-based implementation to the type of problem you are trying to solve, rather than that of the efficiency within the group. And, we should always to attempt to find the best technique within a group for the problem, the best (most efficient) implementation, and the best tool. Since detailed comparisons have not been done within this thesis (but partly in others, e.g. concerning different mutation techniques [124] or search based techniques [150]), we are not dwelling on these details here.

Any of these techniques can be automated in full, but making test case generation efficient is seldom the problem. How efficient the implementation is depends more on the skill of understanding how to implement the technique in code, the set up of the system, and often – for testing – things outside the control, such as build-times of the system, and test environment set up. For most systems the real problem is to have a way to define the outcome, the verdict, of the test. Therefore, the conclusion is that the techniques that use coverage as a measurement, or in a combination with coverage, are always at some level more fair, faster and simpler to determine a measurable improvement for. Our advice is that discussing efficiency of a technique in isolation is not as beneficial for testing on the whole. Instead, one should focus the effort on finding ways to automate as much as possible of the entire test procedure, and have a constant improvement based on our first generic list above. Our contribution here is our general assumption that if techniques are known, there is not a time issue to make a good variable test case instead of a limited test case. Automation takes more time up front, but then execution can be minimized. One has to decide what goals are important for each system under test, and level of test.

14.4 Effectiveness

The effectiveness of TDTs is in principle immeasurable and totally dependent on how much faults that exists in the software system, and is also related to many other items, like the level of detail of the requirement, the process and know-how in the system design, the skill of programming, the tool-support, etc. Thus – assuming "equal" distribution of faults, and "equal" propagation of these faults to the user interface (which already that is a false assumption for real systems), one must conclude that the TDTs are effective in the following order (with the most effective first):

- Negative test
- Boundary Values
- Equivalence Partitioning
- Coverage
- Permutation
- Fault Injection
- Mutation Testing
- Search-Based Testing
- State-Based testing techniques
- Positive testing

Basically, all techniques targeting negative, invalid data are more likely to reveal faults (failures) and target untested areas of the software. This list is basically based on the notion that most industrial systems are developed (and tested) with the aim to "make it work", which will always be a number one priority (and should be). Thus, the problem is rather to identify – which are important and which are not important failures, and that is much more difficult to make a list of, since it varies with the system and likelihood of occurrence. The best approach is to measure the frequency of these odd faults and put that in relation to how likely they are to occur. This approach often ended up in statistical usage testing; where the time to put attention to the system is based on the where the system is used most. The likelihood of finding faults (failures) increases in complex systems for positive TDTs, the more complex and dependent the system is. Thus, positive testing should not be ignored, hence, the priority in sections 14.1.6 and 14.1.7 is the best advice to continually improve the test. For some systems, with a large user base – or potential high availability

interfaces (e.g. internet/web-based), it is motivated to do extensive negative testing. For most systems, this is not the case. Extensive negative testing is seldom motivated for most industrial systems where more obscure techniques (see Chapter 9) will find failures (faults) but are not the primary focus. The conclusion is therefore that one must carefully know which TDTs are used, and then which fault types (failures) that are revealed by these techniques. This will give a specific analysis that might be advisable for similar types of systems. In particular our Chapter 5 contributes with the Fault Analysis that is needed.

14.5 Evaluating your Test Design

One can judge the maturity level based on a few simple aspects of Test Design. We have therefore defined a points-test with questions to answer. The more points for these questions, the more mature your test design is. Sometimes should be interpreted as on some test level.

Question	0	1	2	Total Points
Do you teach your testers TDTs?	No	Sometimes	Yes	
Do you teach your developers TDTs?	No	Sometimes	Yes	
Do you automate your test execution?	No	Sometimes	Yes	
Do you measure coverage?	No	Sometimes	Yes	
Do you fulfill making complete coverage (e.g. at least all code is executed once = 100%)?	No	Sometimes	Yes	
Are you mainly testing positive tests (and using TDTs) that check that things work?	Yes	Sometimes	No	
Are time limits the main reason for not fulfilling test goals?	Yes	Sometimes	No	
Do you use test specifications	No	Sometimes	Yes	

Question	0	1	2	Total Points
(according to standard)?				
Do you define your test cases according to standard?	No	Sometimes	Yes	
Are you specifically checking for common mistakes done in test design, and focus on remedies?	No	Sometimes	Yes	
Do you have an improvement process, which includes that the quality, fault finding ability, and the know-how on the testing is constantly improving?	No	Sometimes	Yes	
Are you randomizing (using automation) on any part of the test case (input selection, path, order of test case etc)?	No	Sometimes	Yes	
Are you experimenting with generation of test cases (and automation of test results) (e.g. state-based, search-based or fault-injection) techniques.	No	Sometimes	Yes	
How many % of your test/development team do you have in the Test Design phase? (One could really measure how many are working in this phase compared to in the test execution phase?)	Few	About 50% of the team	100 % of the team	

14.6 Improving Test Design

The obvious solution is to work on the lower number areas of the above “quiz”. The below description can be viewed as an expansion of the above description, and a solution or priority in test for projects is then dependent on the quality level assumed, this list is based a lot

on the authors 30 years experience with a variety of systems, in addition to the specific studies on these techniques presented in this thesis.

1. Input analysis is made; one would expect that at least one input from each class should be selected – but this seems to be a too large step for many industries. We assume therefore that the analysis is made, but that the usage is expanded over time. This minimal (and sometimes only possible) set is including at least one valid and one invalid input is used based on the technique. The technique works for any system, and is thus always possible to conduct as well as it is possible to use on any level of test, including development/code level.
2. From a "path"/execution step order (flow) view, make sure all normal/default, positive test cases works (includes heuristically selected high priority test cases). This is probably best defined by using a model-based tool. This advice is general, and is valid on any level of test. In fact "TDD" seems to build-up test results by a similar set of test cases.
3. Measure the actual code coverage, and add test cases to achieve 100 % statement coverage. As we noted, this advice is useful for both developers/code-level and functional testers (with code comprehension). Thus, it is sufficient to perform it on one level , and code level comprehension is a must.
    - Note – if there is any dead code, it must be carefully marked, if not removed (it is common to leave dead code as, e.g., instrumentation, or for "future" expansions).
    - Review/code inspects the code that cannot be reached with feasible effort.
4. Complement the test by making sure enough time is spent on different failure-scenarios, and fault behavior through the use of negative TDTs, implicitly this will contribute
5. Make sure you have covered ALL "areas" with TC (also negative test), meaning, and fulfilling equivalence class/category partitioning.
6. All functionality, and all software files/components and managed items should have been tested. Here many often reside into a specific focus of what the requirement specification defines. This

is not sufficient, since there is always a SYSTEM that is delivered, not individual requirements. Thus, one should always prioritize areas with completely new code, since they are likely to contain more faults than code that is untouched and unused. One should be careful about the "software rot" phenomena, where side-effects and lack of attention can make unmodified software that used to work suddenly expose a lot of faults. This is best remedied by thorough analysis of dependability among the software items. Where the reason could be that simple changes in the build-structure, changed and updated hardware (with faster timings) etc., or more commonly a change in the usage of the system.

7. Use Code coverage since branch/decision coverage is not very costly, (assuming 100% statement coverage) this should be completely achieved (for applicable languages). Code coverage to help you identify "holes" in your test and understand the system.
8. If conditions are used – 100% basic condition coverage should be achieved as next step.
9. The final goal for most code is 100% MC/DC coverage (strong mutation coverage). But this is often not feasible other than in a limited "component" view, thus on the small sized code on an implementation level.
10. Permutation of the order of test cases (within limits), or repeat test cases to add to the "execution stress" of the system.
11. In addition, add test cases by generating random input, and random steps or execution order between test cases. At this stage, experimenting with search-based techniques (especially if embedded in a tool) is advisable.
12. Negative approaches are not thoroughly investigated, and a big concern for robustness.
13. System characteristics should always be attempted to be broken down as early as possible, as well as understood and a principle to be designed for in the architecture, early and up-front in the development process. If e.g. usability or performance is poor at system test, it is often too late to totally redesign the system to improve it. Thus, any non-functional requirements are often a combination of exercising a series of functional requirements. Make sure that this is handled in your test design. Particular

advice on these areas is not the focus of this thesis, and further research and understanding must be provided.

We have not discussed organization or roles. But obvious suggestions are to improve and strengthen the test design phase with people with the right skills, knowledge in programming to turn the test cases into code, analyzing skills, and a sufficient deep comprehension on the techniques, but specifically on how to apply the techniques in different situations and on different systems.

14.7 Consequences on Our Test Design Strategy

The result of the above improvements will give you a reasonably high quality. But there might still be other qualities that are missed. Like business aspects that misses to adapt to new business realities on time, or lack the ability to change to fit customer appropriately. It is very common that system characteristics are not properly broken down, thus are only requested up-front, and only measured at the last stages, when the entire system is integrated. This is a risky approach, and can cost a lot of money. Break down of requirements, particularly system requirements are difficult, but efforts must be made to find e.g. performance issues or usability issues up front. This should be a strong design principle, and is often the reason why prototyping is used, to attempt getting an early proof-of-concept in difficult systems, before completing the entire system.

Note that test can only aim to measure and attempt to show what works and what does not, within the given solution. It has not the ability to capture missing problems outside the realm of what is specified (though one can speculate that it seems the analytic abilities of testers often possess this quality too). This step (capturing missing problems) could be found in large systems (where a “next level” would find missing combinations in the level below), and this is often referred to as “faults of omission”. In many complex systems, it seems that what is not clearly defined in the system, is neither described, required nor designed for, and is often outside of the scope of the test and verification. This could more be viewed as the problem of validation, hence validating that the system does what it is supposed

to do. Validation is a different business than testing or verification, and outside the scope of this thesis.

Main problems for test in the above manner are:

1. Maintainability of the test suite becomes too costly compared to the amount of test cases found. Overlap must be reduced, and basic know-how of how the suite executes (non-automatically) must remain in the organization.

2. The test suite must constantly be adding new test cases and find new faults! Tests that does not find faults (but secures e.g. legacy) should be executed only once for every release of the software.

3. Care must be taken to make sure that test cases are finding faults and adding to the coverage.

4. Work should specifically be spent on breaking down system requirements for “earlier” feedback

5. Additional work must be taken to better validate that the system matches the user expectations and business goals. Verification and Test will never assure that the right system is built, and only that the system is built according to its descriptions and requirements and work as described. Understanding these nuances is necessary for the success of a software product.

Chapter 15. Conclusions

15.1 Summary

The main contributions of this thesis are that it

- Defines a way to compare TDTs
- Clarifies the concept of applicability and comprehension of different techniques
- Suggests a way to group TDTs, and provides a taxonomy to clarify overlapping techniques
- Provides guidelines for Industry on test design, with regards to efficiency, effectiveness and applicability

15.2 Expected Impact of this thesis

15.2.1 Impact on Software Industries

The entire software industry can potentially benefit from the results of this thesis. It targets test design, the core of testing – that focus on how we create test cases. It aims to identify ways to compare TDTs, and suggest ways of improving the "fault-fix-retest" correction loop, but by identifying common mistakes that makes test automation a costly step also targets the core of where industries and testers miss in their know-how of TDTs. It describes the importance of different TDTs, and gives a directly deployable guideline. The impact of improving the test cases is expected to lead to

- Higher (test) code coverage and more likely to find failures
- Faster test automation
- Clarification on the importance of different TDTs

- An understanding of when techniques are synonyms, or complementary, including which TDTs conceptually subsumes which other TDTs
- Improved guidance on what are probable and possible mistakes made in test design
- Improved understanding of evaluation of systems through test
- Better understanding of the ways that test design improves efficiency in the test process, through set up and automation

Software is ubiquitous and so are test design and TDTs. The intention was never to limit the result to any particular domain. It seems like one of our results is that they do not differ as much in the Test Design phase. In our guidelines, the scope of suggestions is made as generic as possible. If we do not test, we do not know if the system works, it is as simple as that. Many organizations have yet no clear understanding of these truths about software, and do not understand how much software impacts their product – something that must be understood first and foremost. We find that this basic understanding is a necessity for any software producing company – as important as understanding how quality of the software is directly related to the economics of producing and maintaining it.

15.2.2 Academic Impact

The main task of this thesis is to close the gap between empirical studies in academia, and the need of scalable and useful solutions for industry. In an academic sense, test design and TDTs seems to be a fairly mature science, with seminal work done by Myers [163], Hetzel [98], and Beizer [18] [21] and defined well in Juristo [128], Vegas [203] and Murnane's [159] work. Yet the fact remains that test case construction is still not at the efficiency and effectiveness level it should be in industry, and that applicability is not well understood in academia. In fact, if a method or solution is not applicable, it is plainly not useful. Another aspect of applicability is scalability. A solution might work for some persons in some situation, but if it does not scale across large complex software, and cannot be deployed by the masses, then it is not useful in practice, although the work may have scientific merit. Academia must ask itself, why develop methods and approaches that are not useful for others than the small elite? The

solution must be to identify the gaps between academia and industry and successfully apply scientific measurements, theory and methods to overcome these gaps. This is possibly the main merit of this thesis. Our focus on better understand how the TDTs are used in industry, and how we can improve testing by focusing on the most common misunderstandings in creating test cases using techniques. We hope our clarification on how to better compare and apply these techniques will inspire further research in the area and thus highlight the focus on better industrial deployment of TDTs in the long run. Our research identifies important gaps in research. Some of these are addressed in this thesis and some just put on hold for future investigations. For example, there is a lack of well-defined fault-lists to be injected to evaluate different systems and different domains. Analysis of different test suite executions patterns and how they are related to the test case designs and aspects of fault propagation, trace, debugging and fault-fixing are other aspects in need of more research. Much important research needs to be reworked so it can scale to larger systems and take industrial limitations into account.

15.3 Discussion

During the writing of this thesis it has become clear that the major problem is that research has not solved the basic problem of fault taxonomy, with the fault origin in the code and its propagation into a failure where a test case can reveal it. In fact, until that problem is solved, TDT comparison will always be limited to:

1. The system under test – and its unique set of faults and its unique propagation to failures in this system
2. The amount of test cases already done, and thus specific faults already removed in the process
3. The know-how of the people designing the test cases, thus how well you incorporate the specific technique and deliver an accurate test case applicable for the system

Having stated the current limitations, there are also gains that we can conclude in this research.

Our different measurements show us that efficiency is a minor concern within a specific family in test design, when the technique is

known. In particularly, efficiency should be divided from a test design to a test verdict point of view with regard to a TDT. Know-how of the technique and how to apply it for a series of different ways in a system or across systems seems to be the first hurdle.

Efficiency is different among the groups of test technique families, but rather similar in other aspects. It seems that except initial set-up time needed for more automated techniques (search-based test, random selection, mutation testing, state-based techniques), this time would equal out to the test case generation. Automation of the test execution can be done on any test cases, depending on the tools and system environment that needs to be set up. Efficiency seems to be a measurement that is not as related to the TDT, and other factors influence that. We could conclude that how well the test cases are written seems to impact the cost of automation.

It seems like the key target is the depth of know-how of these techniques. If one knows them – they would be applied much more. We could derive that the lack of knowledge is costly, meaning costly transformation to automation and lack of efficiency and variety in use of techniques must have an impact on the quality of the system. This means that there is a great potential for improvement in industry, and many millions to save.

The similarity of techniques used in industry means that the effectiveness of the test is questionable compared to the effort invested. With the same or very slight more effort, one could be achieving much better effectiveness of the test cases, targeting negative areas, smarter selection of input, using complementary techniques and automatically setting up random selection techniques. This would be utilizing the TDTs to increase impact on coverage and fault-finding ability. We expanded the view of effectiveness to encompass both increased coverage and fault finding ability, and academically – not until a fault taxonomy of different types of faults are established, that could be injected into a system to determine its particular fault pattern and propagation into visible failures, we can really talk about effectiveness of TDTs, but only effectiveness on a particular system. Thus, one must take into account that effectiveness will always be measured in relation to the previous tested parts of the system, and the likelihood of faults propagating into the new and changed parts.

Finally, our contribution on viewing applicability in a new light has brought us new visions on test design. Applicability means not only that the TDT has to be possible to apply on a particular system and level, but also that it is "deployable" in the sense that the technique is known, relatively easy to comprehend and can be applied.

Our research investigations made us focus on the fact that techniques are really applicable on every level – it is a matter of comprehension of the level they are used, and in combination all techniques seem to contribute to the improvement of test design.

The hurdle of applicability is more in our minds than in our systems, but aspects of goals and access got in our way for truly exploring negative test approaches. We conclude that applicability of a technique is not general, if the technique is defined and tailored for a specific system type. Transformations into generally applicable techniques take more effort, and we have only started this work.

Thus, there are more investigations to make with a wider variety of systems to better judge applicability. In our conclusion we must put emphasis on aspects related to test design. Avoiding mistakes in test case design and improving automatic test management seem to be good contributions to our goal to improve the test design and support guidelines for the industry. Test should not be viewed in complete isolation, since the test process is relating to many other development processes.

15.4 Future Work

Based on our contributions, knowledge of TDTs seems to be a major hurdle in the usage of these techniques. How can knowledge be transferred effectively to the industry?

It sounds easy to increase education on TDTs. The fact is that it must start at the universities, where testing should be an obvious part of the curricula in every course. Focus should not only be spend on "organization and management" or "processes" of test, but the actual test case writing using techniques, and providing accurate assumptions and verdicts for the system under test. It is as important as knowing how to code, this knowledge of how to test that the code is fully or partially correct. The simplest would be that for every piece

of code written, the specific test cases for this code piece should be judged with equal weight and know-how. It is necessary to add the notion of scalability to the solutions used at the university, which essentially means more creating, using and testing more complex programs.

We cannot state enough how much academia needs a better fault taxonomy. This is a difficult work, and currently people seem to provide solutions only for very narrow areas and programming languages. A general fault taxonomy would be the key to a lot of the software testing problems, in particularly understanding which faults propagate into the test execution and are possible to catch with different techniques. Secondly, failure patterns based on this fault taxonomy must be established for the system. There are of course a series of other important work related to TDTs that require further study, particularly work on mutation and fault-injection techniques and search-based techniques. Further, the combinations of different families of TDTs are not fully explored. Test Automation should expand from only viewing execution of the system as a target, into test case generation and test verdict automation. The combination of automatic fault localization, automatic debugging and thus build, and automatic regression testing seems to be an important research topic for complex systems. With this, we hope that our contribution of this thesis can lead to savings in the industry and new inspiring research. Software Testing is one of the main activities to ensure quality systems – which most of societies aspects are dependent upon.

List of papers related to this thesis

Eldh, S., Punnekkat, S., Hansson, H.: *Experiments with Component Test to Improve Software Quality***,** International Symposium on Software Reliability Engineering (ISSRE), IEEE, Trollhättan, Sweden, 2007

Eldh, S., Hansson, H., Punnekkat, S., Pettersson, A., Sundmark, D.: *A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques*, Proc. TAIC, IEEE, London, UK, 2006

Eldh, S., Punnekkat, S., Hansson, H., Jönsson., P.: *Component Testing is Not Enough - A Study of Software Faults in Telecom Middleware*, Proc. 19th IFIP International Conference on Testing of Communicating Systems TESTCOM/FATES, Springer LNCS-4581, Tallinn, Estonia, 2007

Eldh, S.; Brandt, J.; Street, M.; Hansson, H.; Punnekkat, S.; *Towards Fully Automated Test Management for Large Complex Systems*, Third International Conference on Software Testing, Verification and Validation (ICST), pp.412-420, 6-10, Paris, France, April 2010

Eldh, S., Punnekkat, S., Hansson, H.: *Analysis of Mistakes as a Method to Improve Test Case Design*, IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), vol., no., pp.70-79, 21-25 Berlin, Germany, March 2011

Thesis

Eldh, S.: *On Evaluating Test design techniques in an Industrial Setting* Licentiate Thesis No. 78, Mälardalen University Press, Västerås, Sweden 2007

Other Papers

Zang. Y., Alba, E., Durillo, JJ., Eldh, S., Harman, M.: *Today/future Importance Analysis*, Proceedings of the 12th annual conference on Genetic and evolutionary computation, ACM, July (2010)

Eldh, S.: *How to Save on Quality Assurance – Challenges in Software Testing*, Jornadas sobre Testeo de Software, p 103--121, ITI, Universidad Politecnica de Valencia, Valencia, Editor(s): Tanja E. J. Vos, 2006

Eldh, S.: *Evaluating Test Techniques using Fault and Failure Analysis*, 7th Conference on Software Engineering Research and Practices in Gothenburg, Sweden, Oct. 2007

Eldh, S.: *Making a Difference with Test Assessment,* Proceedings of the 10th Int. Conference, STARWest, Anaheim, CA, USA, Nov. 2002

Eldh, S.: *Use and Misuse of Software Metrics,* 3rd Int. Conference ASMA, Melbourne, Australia, Nov. 1996

Eldh, S.: *The Challenge of Test Automation – a Process Change*, Proceedings of the 13th International Conference on Testing Computer Software, Washington DC, USA, June 13-16 1996

Eldh, S.: *Reaching Calm from Calamity*, Proceedings of the 11th Int. Conference on Testing Computer Software, Washington DC, USA, June 1994

Eldh, S.: *Automatiserad testing,* (eng. *Test Automation*) Proceedings of Software '94, Den Norske Dataforening, Oslo, Norway 1994

List of Conferences: Keynotes, Workshops and Tutorials related to this thesis

- Eldh, S.: *Fault-Injection at Ericsson,* CREST 13, UCL, London 2011
- Eldh, S.: *Test Myths and Facts,* Czech Test, March 2011
- Eldh, S.: *Mistakes made in Test Design,* SoftTec, Kuala Lumpur, Malaysia, July 2010
- Eldh, S.: *Testing is the Best Work there is*, Bilbao, Spain, Nov 2009
- Eldh, S.: *From Testability to Finding Failures*, Stockholm, Sweden, ICES, Feb 2009
- Eldh, S.: *Closing the Gap between IT-Industry and University,* SoftTec, Kuala Lumpur, Malaysia, July 2009
- Eldh, S.: *Test Design Techniques,* SoftTec, Kuala Lumpur, Malaysia, July 2009
- Eldh, S.: *Test Design Techniques*, Sofia, Bulgaria, June 2009
- Eldh, S.: *Test Design Techniques*, Vienna Test Managers Meet, Vienna, Austria, June 2009

- Eldh, S.: *Test The Test: Approaches to better Quality*, Conquest, Frankfurt, Germany, Sept 2008
- Eldh, S.: *Become a Better Tester*, Conf. On Agility in Testing, New Dehli, India 2008
- Eldh, S.: *Testing is the Best Work there is*, EuroStar, Stockholm, Sweden 2008
- Eldh, S.: *Creating a Company Wide Measurement Program for Verification*, Seoul, International Software Testing Conference (ASTA), South Korea, October 2007
- Eldh, S.: *Achieving Quality – Efficient Verification of Telecom Systems,* ESQAT, Beijing, China, September 2007
- Eldh, S.: *Towards Testing Excellence***,** Proceedings of the I&V Conference, Ericsson, Linköping, September 2005
- Eldh, S.: *Testability – Knowing your tests***,** Proceedings of the 11th Int. Conference EuroStar, Köln, Germany, November 2004
- Eldh, S.: *Software Quality Rank- Improving Designers Test*, Tutorial of the 11th Int. Conference EuroStar, Köln, Germany, November 2004
- Eldh, S.: *Software Quality Rank – An Improvement in Component Test***,** Proc. of the International Conference ICSTest, Dusseldorf, Germany, April 2004
- Eldh, S.: *How to write an effective Test Plan according to IEEE Std 829***,** Proceedings of IIR Conference Testus, Helsinki, Finland, March 2003
- Eldh, S.: *To be a Leader in Testing***,** Proceedings of IIR Conference Testus, Helsinki, Finland, March 2003
- Eldh, S.: *Test Assessments based in TPI***,** Proceedings of the 10th Int. Conference EuroStar, Edinburgh, Scotland, November 2002
- Eldh, S.: *Software Testing Issues and Challenges from an Ericsson Perspective***,** Seminar, ARTES, Stockholm, Sweden, August 2002
- Eldh, S.: *A Successful TPI Assessment***,** Proceedings of the Advanced Testing Conference, Helsinki, Finland, March 2002
- Eldh, S.: *How to improve Dynamics in a Test Team***,** Proceedings of the Advanced Testing Conference, , Helsinki, Finland, March 2002
- Eldh, S.: *From Art of Software Testing to Leadership***,** Closing Keynote of the 9th Int. Conference EuroSTAR, Stockholm Sweden, 2001
- Eldh, S.: *Successful Test Strategies***,** Proceedings of the 9th Int. Conference EuroSTAR, Stockholm, Sweden 2001
- Eldh, S.: *Measurements in Testing***,** Full day Tutorial, 9th Int. Conference EuroSTAR, Stockholm, Sweden 2001

- Eldh, S.: *Testning förr och nu*, *(eng. Test now and then),* Proceedings of IIR Conference "Testning, Verifiering och Test Strategier", (eng. Testing, verification and test strategies) incl. Workshop on Test Strategies, Stockholm, Sweden, May and Sept 2001
- Eldh, S.: *Testmodeller*, *(eng. Test Processes and models)* Proceedings of IIR Conference "Testning, Verifiering och Test Strategies", (eng. Testing, verification and test strategies) incl. Workshop on Test Strategies, Stockholm, Sweden, May and Sept 2001
- Eldh, S.: *Testdomäner, (eng. Test domains),* Proceedings of IIR Conference "Testning, verifiering och Test Strategies", (eng. Testing, verification and test strategies) incl. Workshop on Test Strategies, Stockholm, Sweden, May and Sept 2001
- Eldh, S.: *Testning i framtiden,* Proceedings of IIR Conference "Testning och verifiering och Test Strategies", (eng. Testing, verification and test strategies) incl. Workshop on Test Strategies, Stockholm, Sweden, May and Sept 2001
- Eldh, S.: *Test Metrics - To measure Test*, SAST Conference, Stockholm, Sweden, Feb 2001
- Eldh, S.: *Why is automatic testing important today*: Proceedings of Sweden Engineering Industries, (Verkstadsindustrier), Conference on "Automatic testing with Daily Build", Stockholm, Sweden, Nov 2000
- Eldh, S.: *Performance Testing*, Proceedings of the 7th Int. EuroStar, Barcelona, Spain Dec 1999
- Eldh, S.: *What is Next in testing*, Keynote of the 7th EuroSTAR, Barcelona, Spain 1999
- Eldh, S.: *Developers in testing* Keynote of the 7th EuroSTAR, Barcelona, Spain 1999
- Eldh, S.: *Advanced Testing in Telecommunications,* 5th Int. EuroSTAR, November, Edinburgh, Scotland 1997
- Eldh, S.: *Essentials in OO-testing,* Proceedings of the 4th Int. EuroSTAR, Amsterdam, Netherlands, Dec 1996
- Eldh, S.: *Helping Developers Do it better in an OO environment*, Proceedings of the 3rd Int. EuroSTAR, London, U.K, also Proceedings of the 5th Int. STAR, 2-5 May 1996, Orlando, Florida, USA, Dec 1995
- Eldh, S.: *Testing with Third Party Products*, Proceedings of the 3rd Int. EuroSTAR, London, U.K, Dec 1995
- Eldh, S.: *Hur görs en ändring i praktiken - de sex faserna,* (eng. How is a change made in practice – the six phases) Proceedings of

”Systemförvaltning för ökad kvalitet”,(eng. System maintenance to increase Quality) (NFI) Stockholm, Sweden, Mars 1993

Book

Eldh, S.: *Operativsystem och datorsystem*, (eng. Operating systems and computer systems) Studentlitteratur, ISBN 91-44-24131-3 Lund, 1987

Master Theses

Contributing Master Theses supervised by S. Eldh, who in addition to normal supervision, created thesis work proposals, approach and context.

- Larsson, M.: *Evaluation and overview of test techniques and their applicability in different real-time systems* IDE, MDH 2004
- Stenmark, J., Bokvist, H.: *Analysis and evaluation of the method Software Quality Rank on component test* IDE, MDH 2004
- Albertsson, D., Sandberg, P.: *An Evaluation Model for Selecting Test Tools*, IDE, MDH 2005
- Jönsson, P.: *Analysis and Classification of Faults and Failures in a Large Complex Telecom Middleware System* IDE, MDH 2007
- Gollapudi, A., Ojha, A.: *Comparing Applicability for Telecom Systems*, MDH 2009
- Di Campli, G., Ordine, S.: *Comparing Test Design Techniques for Open Source Systems*, MDH 2009

References

[1] Age Calculator see online): http://www.Planet-Source-Code.com/ xq/ASP/ txtCodeId.2778/lngWId.2/qx/vb/scripts/ShowCode.htm

[2] Agrawal, H.: "Towards automatic debugging of computer programs", Thesis, Purdu University, 1991

[3] Andrews, J.H., Briand, L.C., Labiche, Y.: "Is Mutation an Appropriate Tool for Testing Experiments?", ICSE'05, St. Louis, Missouri, USA. ACM, 2005

[4] Ammann, P., Offutt, J.: "An Introduction to Software Testing", Cambridge University Press, 2008

[5] Apache see (online): http://www.apache.org/

[6] Apiwattanapong, T., Santelices, R., Chittimalli, P.V., Orso, A., Harrold, M.J. Tata: "MaTRIX: Maintenance-Oriented Test Requirements Identifier and Examiner", Proc. From TAIC, IEEE 2006

[7] Auguston, M., Jeffery, C., Underwood, S., "A Framework for automatic debugging", Proc. of 17'th Int. conf ASE, IEEE, 2002

[8] Avižienis, A., Laprie. J.: "Dependable computing: From concepts to design diversity". Proceedings of the IEEE, volume 74, pages 629–638, May 1986

[9] Ball, T.: "The Concept of Dynamic Analysis", LNCS, Software Engineering — ESEC/FSE '99 , Volume 1687, pp216-234, 1999

[10] Banksystem see online http://www.planet-source-code.com/vb/scripts/ ShowCode. asp?txtCodeId =3994&lngWId=2

[11] Basili, VR. Rombach, HD., "The TAME project: Towards improvement-oriented software environments", Trans. of Softw. Eng. June Vol 6. 1988

[12] Basili, V.R., Perricone, B.T.:"Software errors and complexity: An empirical investigation", Communications of the ACM, 27(1):42–52, 1984

[13] Basili, V., Elbaum S.: "Empirically driven SE research: state of the art and required maturity from "Better Empirical Science for Software. Engineering", Invited Presentation, Shanghai, China, ICSE 2006

[14] Basili, V.R. , R.W. Selby, "Comparing the Effectiveness of Software Testing Strategies" original 1985, revised Dec. 1987, in B. Boehm, H.D. Rombach, Martin V. Zelkowitz (Eds.). Foundations of Empirical Software Engineering, The Legacy of Victor R. Basili, Springer Verlag, 2005

[15] Basili, V.R., Boehm, B.,"COTS-Based Systems Top 10 list", IEEE Computer, Vol. 34, Issue 5, 2001

[16] Beck.K. et al, ”Agile Manifesto” see online at http://www.agilemanifesto.org/
[17] Beck, K., “Extreme programming eXplained: embrace change”, Addison-Wesley, Reading , 2000
[18] Beckman, K., Coulter, N., Khajenoori, S., Mead, N.R.: "Collaborations: closing the industry-academia gap," Software, IEEE , vol.14, no.6, pp.49-57, Nov/Dec 1997
[19] Beizer, B.: Software Testing Techniques, VNR, International Thomson Computer Press, 2nd ed. Boston, 1990
[20] Beizer, B.: “Software Testing and Quality Assurance”, VNR electrical/computer science and engineering series. NY, 1984
[21] Beizer, B.: “Black-box Testing” John Wiley &Sons, ISBN 0-471-12094-4, 1995
[22] Bejeweled see online http://www.planet-source-code.com/vb/scripts/ ShowCode. asp?txtCodeId=5259&lngWId=10
[23] Berner, S. Weber, R. and Keller, R.K. “Observations and Lessons Learned from Automated Testing”, IEEE Proc. of ICSE’05, 2005
[24] Bertolino, A.: “Software Testing Research: Achievements, Challenges, Dreams”, In 2007 Future of Software Engineering, Int. Conf. on Software Engineering. IEEE Computer Society, Washington, DC, pp. 85-103., 2007
[25] Black, R.: “Critical Testing Process”, ISBN 0-201-74868-1, Addison-Wesley, 2002
[26] Black, R.: “Managing the Testing Process”, ISBN 0-471-22398, Wiley publishing Inc., 2002
[27] Boehm, B. W.: “Software Engineering Economics”, Prentice Hall, 1981
[28] Bomberman see online: http://users.ugent.be/~jpwinne/bombman.html
[29] Briand, L. C., Pfal, D. “Using Simulation for Assessing the Real Impact of Test Coveravge on Defect Coverage”, IEEE Transactions on Reliability, Vol 49, No. 1, March 2000
[30] BS 7925-2 British Computer Society, BSI, Swindon, UK, Aug. 1998
[31] Bruno, M., Canfora, G., Di Penta, M., Esposito, G., and Mazza, V.: “Using Test Cases as Contract to Ensure Service Compliance across Releases”, Springer Verlag, LNCS Vol. 3826, pp.87-100, 2005
[32] Buddi system see online: http://buddi.digitalcave.ca/index.jsp
[33] Burnstein, I., Suwannasart, T., Carlson, C.R.: “Developing a Testing Maturity Model: Part II, Proc. Crosstalk”, The Journal of Defense Software Engineering, 1996

[34] Buwalda, H., Janssen, D., Pinkster, I.: "Integrated Test Design and Automation", ISBN 0-201-73725-6, Pearson Ed. Ltd, Addison-Wesley, 2002
[35] Causevic, A., Sajeev, A., Punnekkat, S., "Redefining the role of testers in organisational transition to agile methodologies", International Conference on Software, Services & Semantic Technologies, Oct, 2009
[36] Chen, T., Bai, A., Hajjar, A.: "Fast Anti-Random (FAR) Test Generation to Improve Quality of Behavioral Model Verification", Jour. Electronic Testing: Theory and Application 18, Kluwer, Dec. 2002, pp. 583-594
[37] Chillarege, R., Inderpal S. Bhandari, J.k.. Chaar, M.J., Halliday, Moebus, D.S,,Ray, B.K., and Wong, M-Y.: "Orthogonal defect classification – a concept for in-process measurements", IEEE Trans. on Soft. Eng., 18(11):943–956, 1992
[38] Clean Sheet see (online): http://csheets.sourceforge.net/
[39] Clover see (online): http://www.atlassian.com/software/clover/ CloverDownload Center.jspa
[40] Code Cover see (online): http://www.codecover.org/documentation/ install.html
[41] CodeproAnalytiX see (online): http://www.instantiations.com/codepro/ analytix/download.html
[42] Copeland, L.: "A Practitioner's Guide to Software Test Design", ISBN 1-58053-791-X, Artech House Publishers, 2004
[43] Craig, R.D., Jaskiel, S.P.: "Systematic Software Testing", ISBN 1-58053-508-9, Artech House Publisheing, 2002
[44] Dahl, O-J., et al. (Dijkstra): "Structured programming", Academic Press, London and New York, 1972
[45] Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Patton, G.C., Horowitz, B.M: "Model-based Testing in Practice", Proc. of ICSE, IEEE, 1999
[46] Damm, L.O, Lundberg, L., Wohlin, C.: "Fault-Slip Through - a concept for measuring the efficiency of the test process", Journal of Software Process: Improvements and Practice: Volume 11(issue 1) John Wiley and Sons, pp. 47-59, 2006
[47] DeMillo, R., Keynote at IEEE Workshop on Mutation Testing and Analysis, at TAIC PART, Windsor, September 2007
[48] DeMillo, R.A. Guindi, D.S. McCracken, W.M. Offutt, A.J. King, K.N., **"**An extended overview of the Mothra software testing environment", Proc on Software, Testing, Verification and Analysis, IEEE, July 1988

[49] DeMillo, R.A., Lipton, R.J., Sayward, F.G.: "Hints on Test Data Selection: Help for the Practicing Programmer", IEEE Computer, Vol 11, Issue 4, pp 34 – 41, 1978
[50] DeMillo, R. A. Lipton, R. J. and Sayward, F. G.: "Program mutation: a new approach to program testing", State of the Art Report, Soft. Testing, Volume 2: Invited Papers, Infotech Int., 1979
[51] DeMillo, R.A., Maihur, A.P.: "A grammar based fault classification scheme and its application to the classification of the errors of TEX", Technical Report SERC-TR-165-P, Purdue University, West Lafayette, IN 47907, 1995
[52] DeMillo, R.A., Mathur, A.P., Wong, W.E., Frankl, P.G., Weyuker, E.J.; "Some critical remarks on a hierarchy of fault-detecting abilities of test methods [and reply]", IEEE Trans. on Software Engineering, Vol. 21, Issue 10, pp.858 – 863, Oct 1995
[53] DeMillio, R. A., Offutt, A. J.: "Constraint-Based Automatic Test Data Generation", IEEE Transactions on Software Engineering, vol. 17, no. 9, pp. 900-910, September, 1991
[54] Dijkstra, E.J."Structured Programming", North Atlantic Treaty Organization Committee, 1969, in ACM Book: "Classics in Software Engineering". ISBN:0-917072-14-6 Yourdon Press, NJ, USA 1979
[55] DO-178B/ED-12B "Software considerations in airborne systems and equipment certification", RTCA SC-167, EUROCAE WG-12, Dec 1992
[56] Draw see (online): http://sourceforge.net/project/showfiles.php?groupid=163136
[57] Duran, J.W., Ntafos, S.C., "Evaluation of Random Testing", Transaction of Software Engineering, Vol. 10(44), IEEE, 1984, pp. 438-444
[58] EclEmma see (online): http://www.eclemma.org/installation.html
[59] Eclipse, see (online): http://www.eclipse.org/
[60] Eickelmann, N.S. and Richardson, D.J.: "An Evaluation of Software Test Environment Architectures", IEEE Proc. of ICSE-18, 1996
[61] El Emam, K.: "The ROI form Software Quality", Auerbach publications, Taylor & Francis group, 2005
[62] Eldh, S.: "How to Save on Quality Assurance – Challenges in Software Testing", Jornadas sobre Testeo de Software, p 103--121, ITI, Universidad Politecnica de Valencia, Valencia, Editor(s): Tanja E. J. Vos, 2006
[63] Eldh, S.: "On Evaluating Test Techniques In An Industrial Setting", Mälardalen Uni. Lic. Thesis No.78, 2007

[64] Eldh, S., Punnekkat, S., Hansson, H.: "Experiments with Component Test to Improve Software Quality", International Symposium on Software Reliability Engineering (ISSRE), IEEE, Trollhättan, Sweden, 2007

[65] Eldh, S., Hansson, H., Punnekkat, S., Pettersson, A., Sundmark, D.: "A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques", Proc. TAIC, IEEE, London, UK 2006

[66] Eldh, S., Punnekkat, S., Hansson, H., Jönsson., P.: "Component Testing is Not Enough - A study of Software Faults in Telecom Middleware", Proc. 19th IFIP International Conference on Testing of Communicating Systems TESTCOM/FATES, Springer LNCS-4581, Tallinn, Estonia 2007

[67] Eldh. S.: "Reaching Calm from Calamity", Proc. of the 11th Int. Conference on Testing Computer Software, Washington DC, June 13-16 1994

[68] Eldh, S.: Hansson, H.: Punnekkat, S.: "Analysis of Mistakes as a Method to Improve Test Case Design," Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, vol., no., pp.70-79, 21-25 March 2011

[69] Eldh, S.: Brandt, J., Street, M., Hansson, H., Punnekkat, S.: "Towards Fully Automated Test Management for Large Complex Systems," Software Testing, Verification and Validation (ICST), 2010 Third International Conference on , pp.412-420, 6-10 April 2010

[70] Eldh, S., Hansson, H., Punnekkat, S.: "Analysis of Mistakes as a Method to Improve Test Case Design," Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, vol., no., pp.70-79, 21-25 March 2011

[71] El-Far, I. K., Whittaker, J. A.: "Model based Software Testing", Appears in 2010 This paper appears in the Encyclopedia on Software Engineering (edited by J.J. Marciniak), Wiley, 2001

[72] Emanuelsson, P., Nilsson, U.: "A Comparative Study of Industrial Static Analysis Tools", Electronic Notes in Theoretical Computer Science, Volume 217, 21 July 2008

[73] Endres, A.: "An analysis of errors and their causes in system programs". Technical report, IBM Laboratory, Boebligen, Germany, 1975

[74] EuroBudget see (online): http://sourceforge.net/project/showfiles.php?group_id=48615

[75] Fagan, M. E.: "Design and Code Inspection reduce errors in program development", IBM Journal No 3, 1976

[76] Failure Tracking tools see (online): http://www.testingfaqs.org/t-track.html

[77] Fenton, N., Pfleeger, S.L.: “Software Metrics a Rigours Approach”, PWS Publishing, 1997
[78] Fewster, M. Graham, D.: “Software Test Automation”, ISBN 0-201-33140-3, ACM Press, Addison-Wesley, 1999
[79] Frankl P. G., Iakounenko, O.: “Further Empirical Studies of test Effectiveness”, Proc. 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Orlando (FL, USA) 153-162 1998
[80] Gao, J. Z., Itaru, F. and Touoshima, Y. ”Information Technology and Management” 3, 85–112, Kluwer Academic Publishers, 2002
[81] Ghazarian, A.: “A Case Study of Defect Introduction Mechanisms”, LNCS, Volume 5565/2009, pp 156-170, 2009
[82] Ghazarian, A.: "A Case Study of Source Code Evolution," Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on , vol., no., pp.159-168, 24-27 March 2009
[83] Giruado, G., Tonella, P. and Basili, V. (ed). “Designing and Conducting an Empirical study on Test Management Automation”, Empirical Software Engineering, 8, Kluwer Academic Publishers, p. 59–81, 2003
[84] GNULinux Linux, see (online): http://www.gnu.org/
[85] Gollapudi, A., Ojha, A.: “Comparing Applicability for Telecom Systems”, Master Thesis, Mälardalen University, 2009
[86] Goodenough, J.B., Gerhart. S.L.: “Toward a theory of test data selection”. Proc. of the int. Conf. on Reliable software, pages 493–510, New York, NY, USA, ACM Press. 1975
[87] Graham, D.R., Herzlich, P., and Morelli, C., “CAST Report - Computer-Aided Software Testing.” Cambridge Market Intelligence, Limited, London, House, Parkgate Road, London, UK, 1995
[88] Gray. J.: “Why do computers stop and what can be done about it?” Technical Report 85:7, Tandem Computers, 1985
[89] Greenwood, D.J. and Levin, M. “Introduction to Action Research: Social Research for Social Change” (2nd Ed.), Thousand Oaks, California, Sage, 2006
[90] Halloran, T. J., Scherlis, W. L,. “High Quality and Open Source Software Practices” 2nd WS on Open Source Software, 2002
[91] Halstead, M.H.: “Elements of Software Science.” Elsevier North-Holland, Inc. .ISBN 0-444-00205-7, Amsterdam, 1977
[92] Hamlet, D., Taylor, R.: “Partition testing does not inspire confidence [program testing]” Transaction of Software Engineering, Vol.16(12), IEEE, Dec. 1990

[93] Hardie, Fred H., Suhocki, Robert J.: "Design and Use of Fault Simulation for Saturn Computer Design," Electronic Computers, IEEE Transactions on , vol.EC-16, no.4, pp.412-429, Aug. 1967

[94] Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A.: "Regression test selection for Java software". SIGPLAN Not. 36, 11 pp. 312-326, Nov. 2001

[95] Harrold, M. J., Offutt, A. J., Tewary, K.: "An approach to fault modelling and fault seeding using the program dependence graph", Journal of Systems and Software, 36(3):273–296, March 1997

[96] Hatton, L.: "Safer C: developing for High-Integrity and Safety-Critical Systems", McGraw-Hill, 1995

[97] Henningsson, K., Wohlin, C.: "Assuring fault classification agreement – An Empirical Evaluation", Proc. of ISESE'04, IEEE 2004

[98] Hetzel, W.C.: "An experimental analysis of program verification methods", PhD dissertation, Univ. North Carolina, Chapel Hill 1976

[99] Hetzel, W.C: "The complete guide to software testing", ISBN 0-471-56567-9 John Wiley & Sons 1984

[100] Hierons, R. Brunel University, Keynote speech at TTCN-3 conference, Stockholm, Sweden, June 2007

[101] Howden, W.E.: "Reliability of the path analysis testing strategy" IEEE Trans. on Software Engineering, 2(3):208–215, Sept. 1976

[102] HP Test Director see (online): https://h10078.www1.hp.com/ cda/ hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24%5E1131_4000_100__

[103] Humphrey, W.: "Managing the Software Process", Addison Wesley Professional, MA 1989

[104] Hutchins, M., Foster, H., Goradia T, Ostrand, T.: "Experiments of the effectiveness of dataflow- and control flow-based test adequacy criteria". In Proc. of the 16th int. conf. on Software engineering, pages 191–200, IEEE Computer Society Press. Italy, 1994.

[105] Hyunsook, D., Elbaum, S., Rothermel, G.: "Infrastructure support for controlled experimentation with software testing and regression testing techniques", Proc. Int. Symp. On Empirical Software Engineering, ISESE '04, pp. 60 – 70, ACM Aug. 2004

[106] IBM Jazz Technological Platform, see (online): http://www-01.ibm.com/ software/ rational/jazz/

[107] IBM Rational® Purify see (online): http://www-01.ibm.com/software/awdtools/purify/

[108] IBM Rationa® Quality Manager see (online): http://www-01.ibm.com/ software/awdtools/rqm/standard/
[109] IEC 61508 Std Functional Safety, 2nd ed. IEC SC (Subcommittee) 65 A: Industrial-process measurement, control and automation - Systems aspects, 2010
[110] IEEE Std. 829 Standard for Software Test Documentation -1998 & 2008
[111] IEEE Std. 1044 -1993 Standard for classification for software anomalies, 1993
[112] ImageJ see (online): http://rsbweb.nih.gov/ij/index.html
[113] Image Processing see (online): http://www.planet-source-code.com/vb/ scripts/ShowCode.asp?txtCodeId=5869&lngWId=10
[114] IRClient see (online): http://www.planet-source-code.com/ vb/scripts/ ShowCode.asp?txtCodeId=810&lngWId=10
[115] Imbus TestBench, see (online): http://www.imbus.de/products/ imbus-testbench/
[116] Ishoda, S. "A criticism on the capture-and-recapture method for software reliability assurance", Proc. Soft. Eng. IEEE, 1995
[117] ISO/IEC 17799, Information Technology - Code of Practice for Information Security Management, 2002
[118] ISO/IEC 25000 Int. Std. "Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE ISO/IEC 25000:2005(E), 2005
[119] ISO/IEC 9126-1:2001 "Software engineering — Product quality", ISO/IEC Std, 2001
[120] Itkonen, J., Mantyla, M. V. , Lassenius, C.: "Defect Detection Efficiency: Test Case Based vs. Exploratory Testing,", First International Symposium on Empirical Software Engineering and Measurement pp.61-70, 2007
[121] Javasusp see (online): http://sourceforge.net/scm/?type=cvs&group_id =215369
[122] JMoney, see (online): http://sourceforge.net/project/showfiles.php? groupid=17482
[123] Jedlitschka, A., Pfahl, D.: "Reporting guidelines for controlled experiments in software engineering," Empirical Software Engineering, 2005. 2005 International Symposium on , pp. 10 pp., 17-18 Nov. 2005
[124] Jia, Y., Harman, M.: "An Analysis and Survey of the Development of Mutation Testing," IEEE Transactions on Software Engineering, vol. 99, 2010
[125] Johnson C. et. Al.: "Guide to IEEE standard for classification for software anomalies", Technical report, IEEE Computer Society, March 1995

[126] Jones, B.F. Sthamer, H.-H. Eyres, D.E.: "Automatic Structural testing using genetic algorithms", Journal of Software Engineering, IEEE, Sept 1996

[127] Jorgensen, P. "Software Testing: A Craftsman's Approach", Department of Computer Science and Information Systems, Grand State University, Allendale, CRC Press, 1995

[128] Juristo, N., Moreno, A.M. and Vegas, S.: "Reviewing 25 Years of Testing Technique Experiments", Journal of Empirical Software Eng., Springer Issue: Volume 9, Numbers 1-2, pp. 7 – 44, March 2004

[129] Juristo, N., Moreno, A.M. and Vegas, S.: "Limitations of Empirical Testing Technique Knowledge", Lecture Notes on Empirical Software Engineering archive, pages 1-38. World Scientific Publishing Co., Inc. River Edge, NJ, USA, 2003

[130] Jönsson, P.: "Analysis and Classification of Faults and Failures in a Large Complex Telecom Middleware System", IDE, MDH, Master Thesis, Nov 2007

[131] Kamsties, E., and Lott, C. M.: "An empirical evaluation of three defect-detection techniques", Proc. Fifth European Soft. Eng. Conf., (ISERN), Spain 1995

[132] Kaner, C. Falk, J. Nguyen, H.Q.: "Testing Computer Software", 2nd ed., VNR, International Thomson Computer Press, 2nd ed. Boston, 1993

[133] King, J. C.: "A new approach to program testing". In Proc- of the Int. Conf. on Reliable Software (Los Angeles, California, April 21 - 23, ACM, New York, NY, 228-233. 1975

[134] King, J. C.: "Symbolic execution and program testing", Commun. ACM 19, 7 (Jul. 1976), 385-394. 1975

[135] Kit, E.: "Software testing in the real world: improving the process", ISBN 0-201-87756-2, ACM Press, Addison-Wesley, 1995

[136] Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J.,: "Preliminary guidelines for empirical research in software engineering," Software Engineering, IEEE Transactions on , vol.28, no.8, pp. 721- 734, Aug 2002

[137] Knuth, D.E.: "The errors of TEX" Software Practice and Experience, (7):607–685, 1989

[138] Koomen, T., Pol, M.: "Test process improvement: a practical set-by-step guide to structured testing", Addison-Wesley Longman, 1999

[139] Larsen, L. and Harrold, M. J.: "Slicing object-oriented software", In Proceedings of the 18th international Conference on Software Engineering (Berlin, Germany, March 25 - 29, 1996.

[140] Larsson, M.: “Evaluation and overview of test techniques and their applicability in different real-time systems”, IDE, Mälardalen University, 2004
[141] LaTeXDraw see (online): http://latexdraw.sourceforge.net/
[142] Legeard, B., Peureux, F., Utting M.: “Automated boundary testing from Z and B”, Lecture Notes in Computer Science, Springer Verlag, 2002
[143] Lewin, K.: “Action research and minority problems”, J Soc. Issues 2(4): 34-46, 1946
[144] Lyu, M.R.: "Software Reliability Engineering: A Roadmap", Future of Software Engineering, 2007. FOSE '07, On page(s): 153 - 170, Volume: Issue: , 23-25 May 2007
[145] Malaiya, Y.K., Li, M.N., Bieman, H.M. Karcich, R.: “Software Reliability Growth with Test Coverage”, IEEE Transactions on Reliability, Vol 51, pp.420-426, Dec 2002
[146] Martin, R. C.: “Chapter 17: Smells and Heuristics - G25 Replace Magic Numbers with Named Constants". Clean Code - A handbook of agile software craftsmanship, ISBN 0-13-235088-2 Boston Prentice Hall, p. 300, 2009
[147] McCabe, T.J., Butler, C.W.: “Design complexity measurements and testing”, Communications of the ACM, 1989
[148] McCabe, T.J.: “A Complexity Measure”, IEEE Transactions on Software Engineering, pp. 308-320, December, 1976
[149] McDonald, M., Musson, R., Smith, R.: “The Practical Guide to Defect Prevention”, ISBN 13 978-81-7853-127-4,WP Publishers & Distributors Private Ltd. 2008
[150] McMinn, P.: “Search-based software test data generation: a survey”. Software Testing, Verification and Reliability, 14: 105–156, 2008
[151] MIL-P-1629 – Procedures for performing a failure mode effect and critical analysis, Department of Defense (US). 9 November 1949
[152] Millner, B.P. Fredrikson, L. and So, B.: “An Empirical Study of the Reliability of UNIX Utility”, Communications of the ACM, 33(12):32-44
[153] Mills, H. D., Dyer, M. and Linger, R.: “Cleanroom Software Engineering”, IEEE Software,**4**(5), 19-25, 1987
[154] Mozilla, see (online): http://www.mozilla.org/
[155] Mockus, A., Fielding, R. T., and Herbsleb, J. D. “Two case studies of open source software development: Apache and Mozilla. ACM Trans. Software Engineering. Methodology. 11, 3, 2002
[156] Mohagheghi, P., Conradi, R., and Borretzen, J. A.: “Revisiting the Problem of Using Problem Reports for Quality Assessment”, ICSE'06,

The 4th Workshop on Software, Quality, WoSQ'06, pp. 45-50, Shanghai, China, 2006.

[157] Murnane, T., Reed, K., Hall, R.: "Towards Describing Black-Box Testing Methods as Atomic Rules", IEEE Proceedings of Australian Software Engineering Conference, ASWEC, 2005

[158] Murnane, T., Reed, K., Hall, R.: "Tailoring Black-box methods", IEEE Proceedings of Australian Software Engineering Conference, ASWEC, 2006

[159] Murnane, T., Reed, K., Hall, R.: "On the Learnability of Two Representations of Equivalence Partitioning and Boundary Value Analysis", IEEE Proceedings of Australian Software Engineering Conference, ASWEC, 2007

[160] Murphy, B., Garzia, M., Suri, N.: "Closing the Gap in Failure Analysis", Workshop on Applied SW Reliability-DSN, (2006)

[161] Müller, T, Black, R., Eldh, S., Graham, D., Olsen, K. Pyhäjärvi, M., Thompson, G., van Veendendal. E, "Certified Tester - Foundation Level Syllabus" - Version 2007, International Software Testing Qualifications Board (ISTQB), Möhrendorf, Germany, 2007.

[162] Myers, G.J.: "A controlled experiment in program testing and code walkthroughs inspections", Comm. ACM 760-768 Sept 1978

[163] Myers, G.: "The Art of Software Testing", 2nd ed. John Wiley & Sons, 1979

[164] MySQL v.5.1 see (online): http://www.mysql.com/

[165] Nebut, C., Fleury, F., Le Traon, Y., Jezequel, J.-M.: "Automatic test generation: a use case driven approach", IEEE Trans. On Soft. Engin. Vol 32, Issue 3, 2006

[166] NIST-Final Report "The Economic Impacts of Inadequate Infrastructure for Software Testing", Table 8-1, National Institute of Standards and Technology, May 2002

[167] Nguyen, D. C., Perini, A., and Tonella, P.: "A Goal-Oriented Software Testing Methodology", Volume 4951/2008, p. 58-72, LCNS, Springer Verlag, 2008

[168] Offutt, A. J., Huffman-Hayes, J.: "A Semantic Model of Program Faults. Proceedings of ISSTA, 195-200, 1996

[169] Offutt, A. J., Rothermel, G., and Zapf, C.: "An experimental evaluation of selective mutation", Proc. the 15th Int. Conf. on Soft. Eng. IEEE, 1993

[170] Opensource website see (online): http://www.opensourcetesting.org/testmgt.php/

[171] Ordine, S., Di Campli, G.: "Comparing test design techniques for open source systems", Master Thesis at Mälardalen University, Sweden Västerås, 2009
[172] Ostrand, T. J. and Balcer, M. J.: "The category-partition method for specifying and generating functional tests**",** Commun. ACM 31, 6, pp. 676-686. June 1988
[173] Ostrand, T. S., and Weyuker, E.: "Collecting and Categorizing Software Error Data in an Industrial Environment", The Journal of Systems and Software, 4:289-300, 1984
[174] Ostrand, T. J., Weyuker, E.J, Bell, R.M.: "Predicting the Location and Number of Faults in Large Software Systems", IEEE Trans. of Soft. Eng., Vol. 31, no 4 April 2005
[175] Paulk, M et al. SEI - Software Engineering Institute: "The Capability Maturity Model: Guidelines for improving the Software Process, Addison-Wesley, 1994
[176] Perry, D.E., Steig, C.S.: "Software faults in evolving a large, real-time system: a study", In 4th European Software Engineering conference – ESEC93, Germany, (1993), 48–67
[177] Pocatilu, P.: "Automated Software Testing Process", 9 Economy Informatics, no. 1, p.97-99, 2002
[178] QAtraq Professional, see (online): http://www.testmanagement.com/
[179] SilkCentral Test Manager, Borland, see (online): http://www. borland.com/us/products/silk/silkcentral_test/index.aspx
[180] QuickCheck see (online): http://www.quviq.com/
[181] Rapps, S. and Weyuker, E. J.: " Selecting Software Test Data Using Data Flow Information", IEEE Trans. Softw. Eng. 11, 4,pp. 367-375, Apr. 1985
[182] Reid, S.C., "Module Testing Techniques – which are the most effective? Results of a Retrospective Analysis", 5th European Int. Conference on Software Testing, Analysis and Review, Edinburgh, November 1997
[183] Robson, C., "Real world research: A resource for social scientists and practitioner-researchers", 2nd ed., Blackwell, (ISBN 0631176888), 1993
[184] Rothermel, G., Elbaum, S., "Test case prioritization: a family of empirical studies, Transaction of Software Engineering, Vol.: 28, Issue 2, page(s): 159-182, Feb 2002
[185] Rothermel, G., Harrold, M. J. "Analyzing regression test selection techniques", IEEE Transaction on Software Engineering, Volume 22, Issue 8, pp. 529-551, August 1996

[186] Runeson, P., Höst, M., “Guidelines for conducting and reporting case study research in software engineering”, Empirical Software Engineering Volume 14, Number 2, 131-164, 2008
[187] Schütz, W: “The Testability of Distributed Real-Time Systems”, Springer Verlag, 1993
[188] Sjöberg, D.I.K., Dybå, T., Jörgensen, M.: “The future of Empirical Methods in Software Engineering Research”, IEEE Future of Software Enginering, (FOSE) 2007
[189] Spillner, A., Linz, T., Shaefer, H.: ”Software Testing Foundations”, ISBN 3-89864-363-8, dpunkt.verlag, Heidelberg, 2005
[190] Stake, R.E.: “The Art of Case Study Research”, Thousand Oaks, California: Sage, 1995
[191] Stamelos, I., Angelis, L., Oikonomou, A., Bleris, GL.: “Code quality analysis in open source software” , Info Systems Journal 12, 2002
[192] Stenmark, J., Bokvist, H.: “Analysis and evaluation of the method Software Quality Rank on component test”, Master Thesis, Mälardalen University, Västerås, 2004
[193] StudentHelper see (online): http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=1570&lngWId=10
[194] Teasley, B., Leventhal, L.M., and Rohlman, S.: “Positive test bias in software testing by professionals: What’s right and what’s wrong”. In Empirical Studies of Programmers: Fifth Workshop, C.R. Cook, J.C. Scholtz, and J.C. Spohrer, Eds. Ablex, Norwood, NJ, 1993
[195] Test Case Template & Additional data and information, regarding Systematic Mistakes in Test Design Study: http://www.idt.mdh.se/~seh01/ICST2011/ , 2011
[196] Thane, H., Wall, A.: “Testing Reusable Software Components in Safety-Critical Real-Time Systems”, vol 1, 1–2. Artech House Publishers, 2002
[197] T-plan see (online): http://www.t-plan.co.uk/
[198] Trienekens, J., van Veenendaal, E., van Solingen, R., Punter, T., van der Swan, M.: “Software Quality from a Business Perspective”, ISBN 90-267-2631-7, Kluwer BedriftsInformatie B.V. Deventer, 1997
[199] UMLet see (online): http://www.umlet.com/
[200] Vaidyanathan, K., Kishor S., Trivedi: “A comprehensive model for software rejuvenation”. IEEE Trans. on Dependable and Secure Computing, 2(2):124–137, 2005
[201] Van Veenendaal, E.: “The Testing Practitioner”, ISBN 90-72194-65-9, Uitgeverij Tutein Noldhenious, 2002

[202] Vegas, S., Basili, V.R., “A characterization schema for Software Testing Techniques’, Journal of Empirical Software Engineering, 10, Springer, pp. 437-466, 2005

[203] Vegas, S., “Characterization Schema for Selecting Software Testing Technique”, PhD Thesis, Universidad Politécnica de Madrid, 2007

[204] Voas, J.M., McGraw, G. Software fault injection: inoculating programs against errors, John Wiley & Sons, Inc. USA, 1997

[205] Vokolos, F. I., Frankl, P. G.: “Empirical evaluation of the textual differencing regression testing technique” Proc. IEEE Int. Conference on Soft. Maint. USA, 44-53 1998

[206] Wegener, J. Sthamer, H., Jones, B.F. and Eyres, D.E., “Testing real-time systems using genetic algorithms”, Journal of Software Quality, Springer, Vol. 6 (2), June 1997

[207] Weyuker, E.J. and Jeng, B.: ”Analyzing partition testing strategies,” IEEE Trans. Software Engineering, pp. 702-711, July 1991

[208] Weyuker, E.: “An Empirical Study of the Complexity of Data Flow Testing”. Proceedings 2nd Workshop on Software Testing, Verification and Analysis. pp. 188—195. Banff, Canada, 1988

[209] Weyuker, E.J.: “The Cost of Data Flow Testing: An Empirical Study” IEEE Transactions on Software Engineering. Volume 16 (2). pp 121—128, 1990

[210] Whittaker, J.: Invited Keynote at European Int. Conference on Software Testing, Analysis and Review, Copenhagen 2000

[211] Whittaker, J. A.: “How to Break Software: A Practical Guide to Testing”, Addison-Wesley, 2003

[212] Whittaker, J. A., Thomason, M.G., “A Markov Chain Model for statistical software testing.” Transactions on Software Engineering, VOL. 20, No. IO, Oct 1994

[213] Wikstrand, G., Feldt, R., Gorantla, J., Zhe, W., and C. White. “Dynamic regression test selection based on a file cache an industrial evaluation”. ICST, 2009

[214] Williams, A.W., Probert, R.L.:"A practical strategy for testing pair-wise coverage of network interfaces," Software Reliability Engineering, International Symposium on, p. 246, The Seventh International Symposium on Software Reliability Engineering (ISSRE '96), 1996

[215] Wohlin, C., Runesson, P., Höst, M. Ohlsson, M. C., Regnell, B., Wesslén, A.: “Experimentation in Software Engineering: An Introduction”, Kluwer, 2000

[216] Wood, M., Roper, M., Brooks, A., and Miller, J.: "Comparing and combining software defect detection techniques: A replicated empirical study", Proc. of the 6th European Soft. Eng. Conf., Switzerland, 1997

[217] Xu, D. and Xu, W.: "State-based incremental testing of aspect-oriented programs". In Proc. of the 5th int. Conf. on Aspect-Oriented Software Development (Bonn, Germany, March 20 - 24, 2006). AOSD '06. ACM, New York, NY, 180-189, 2006

[218] Yin, R.K.: "Case study research: Design and methods", Third Edition, Sage Publications, Thousand Oaks, CA, (2nd Edition, 1994), 2003

[219] Zeil, S.J.: "Perturbation techniques for detecting domain errors", IEEE Transactions on Software Engineering, 15(6):737–746, June 1989

[220] Zeller, A: "Why Programs FAIL, A guide to Systematic Debugging", dpunkt.verlag, Elsevier Inc. 2006

[221] Zhu, H., Hall, P. A. V., May, J. H. R.: "Software Unit Test Coverage and Adequacy", ACM Computing Surveys, Vol. 29, No. 4 Dec 1997

Appendix 1 Test Case Template & Test Record

(note: please name file with your name & a TC number)

Test case ID: *<number>*	Date: *<2009-11-24>*	Version: *<number>*	Author: *<name>*	Reviewed by: *<name>*
Used in: *<system name>*	System version: *<number>*	Test environment: *<reference>*	Test Suite: *<number>*	Automated *<yes/no>*
Time for TC creation; *<minutes>*	TDT used: *< name >*	Assumptions:		
Starting point:				
Test case:				
Pre-condition:				
Step 1:				
Step 2:				
Input Data (valid)				
Input Data (invalid)				
Output/visible result for passed: (post-condition)				
Side-effect (clean-up)				
Comment:				

Model, flow-chart, visible description (if any):

Test Record

TR for Test case ID: <*number*>	Date: <*2009-11-24*>	Executed Version: <*number*>	Tester:
System version: <*number*>	Test environment: <*reference*>	Test Suite: <*number*>	Automated <*yes/no*>
Result: <*passed/failed/blocked/passed with work-around*>			
Side-effect:			
Time for execution: <*minutes*>	Failure ID nr:	Failure report date: <*2009-11-24*>	Reported to: <*System*
		Reported by: <*name*>	
Failure description:			
Other:			
Comments:			

Note: Only ONE TEST CASE & record per PAGE(S) and FILE!

Appendix 2 TDT Applicability

Lab 1: TDT Applicability (*Note: content edited for thesis format)*

Please follow the instructions below in DETAIL.

We are conducting a research experiment based on this data, and we want the data to be as accurate and precise as possible, but most of all – completely honest. There is no judging put into this, since we believe people use and understand things very differently, and we would like you to honor your personal difference. Our goal with this experiment is to test the applicability of different test design techniques (TDT). We would therefore like you to fill in accurate time and give grades for each part of process as advised. The software used is an open source project. You will have access to the code & the interface.

Please answer the following background questions:

Name: xxxx (this will be erased in the research, and is only to check that lab is conducted, and if any questions are raised)

Age: xx Familiarity of Java (Yes/No): yes, Years of programming 1

Times can be recorded in any appropriate time (seconds, minutes, and hours) just be clear!

Grades are given when is 5 most difficult and 1 is least difficult of the attribute asked. Any 1, 2, 3, 4, 5 can be used. Please do NOT use 4.5 etc…

If something is not possible – or your give up, write N.A. (Not applicable), and formulate WHY in a separate paper (X in comment), please record time until you gave up in this sheet.

Lab 1: Input techniques – creating tests cases!

Design and execute one unique test case for each level you identify: a) code b) integration and c) system (interface) and document each test case, including the verdict (pass, fail). If any defect was found, please document where and what test case (or action) you did to find it. Please do not "attempt" any other tests and try to follow the test as simple as possible.

Last thing, if you do NOT do the questions in order (from upper left – to right, and then downwards) please NOTE the ORDER you conducted them, clearly.

In all you will hand in this sheet, and a documentation of 15 Test Cases and possible some faults and maybe comments to have completed the task Put your name on each.

TDTNR	Test Design Tech. and level	Time to under-stand The TDT and exercise	Time to identify Unique part of where to apply	Grade difficulty to identify where to apply	Time to create a unique Test Case	Grade difficulty of creating Test Case	Estimate (or count) Possible TC you locate for selected target	If fault was found put ref and doc	Comment
		A	B	C	D	E	F	G	H
1	Random System	5 h	120 m	4	1 h	5	1	Fail	
2	Random Integr.	4 h	40 m	5	30 m	4	1	Fail	
3	Random Code	2 h	30 m	4	30 m	4	1	Fail	
4	Normal case, System	2 h	30 m	3	25 m	3	1	Pass	
5	Normal Case, Integr.	3 h	20 m	3	20 m	2	1	Pass	
6	Normal Case, Code	2 h	25 m	3	20 m	4	1	Pass	
7	Faulty TC, System	3 h	50 m	4	35 m	3	1	Fail	
8	Faulty TC Int.	3 h	45 m	4	30 m	3	1	Fail	
9	Faulty TC Code	3 h	35 m	3	35 m	4	1	Fail	
10	Eq. Part, System	2 h	40 m	3	25 m	3	1	Pass	
11	Eq. Part, Integr.	2 h	30 m	4	20 m	2	1	Pass	
12	Eq. Part, Code	1.5 h	20 m	3	10 m	3	1	Pass	
13	BVA,System	1 h	25 m	3	15 m	2	1	Fail	
14	BVA, Integr.	1 h	30 m	3	20 m	3	1	Pass	
15	BVA, Code	1 h	20 m	2	20 m	2	1	Pass	

TDT NR	Test Design Tech. and level	Time to under-stand the TDT and exercise	Time to identify Unique part of where to apply	Grade difficulty to identify where to apply	Time to create a unique Test Case	Grade difficulty of creating Test Case	Estimate (or count) Possible TC you locate for selected target	If fault was found put ref and doc	Comment
		A	B	C	D	E	F	G	H
1									
2									
3									
4									
…									
14									
15									

Lab 2: Other techniques –creating tests cases!

Design and execute at least three unique test cases according to each technique listed

1. State-transition: Draw a model (that will result in one or more test cases)
2. Fault injection (or a single mutation)
3. Exploratory (any random approach)
4. Performance test
5. Any other technique (specify which)

Define WHICH level (Code, Integration, System) you have done the test case(s)! Document each test case, including the expected result and verdict (pass, fail). If any defect was found, please document where and what test case (or action) you did to find it. Please do not "attempt" any other tests and try to follow the test as simple as possible. In all you will hand in this sheet, and a documentation of 15 Test Cases and possible some faults and maybe comments to have completed the task.

(note: instructions same as for Lab 1, removed in thesis, table adjusted).

Appendix 3 Applicability of TDT

(Note: This is edited for space, Variants of this questionnaire did exists scrambling order also specific adaption for Developers/Testers with extra questions)

ID____

1. What is your ROLE (how do you see yourself?) I am most of the time a Developer ___Tester____ Other (what:)_____
2. What is your age? (circle)
   - Less than 25
   - 26 – 30
   - 31 – 35
   - 36- 40
   - 41-45
   - 46 -50
   - 51 – 55
   - 56 and over
3. Do you hold a University Degree? (yes/no)_____
4. How much coding experience do I have (incl. test automation scripting or other) (years): _____
5. What Programming languages do you know and use?
   _____
6. Do you hold a ISTQB or ISEB certification of Testing (yes/no) ____

If yes, do you remember (approximately) when you got certified?_____

7. Years in Testing?_____
8. Years working with Telecom testing?_____
9. Years with testing on other types of system (not telecom) _____
10. If so, what type of system? _____
11. How often do you design completely new test cases (from scratch)?
    _____
12. How often do you change, improve or add on existing test cases?_____
13. How much are your test cases Automated (%)?_____
14. Do you write your test cases before you write the code? (Circle): Always, sometimes, seldom, never
15. Do you delete code that you have no test cases for? (Circle): Always, sometimes, seldom, never
16. Do you know if your test cases that you use are effective (meaning: finding failures/problems easy? (Yes, maybe, no) Please elaborate:
    _____
    _____

ID____

Which if the following Test Design Techniques do you feel familiar with and know how to use? (Give a value 1 to 5, where 1 is I know this really well, and 5 I have never heard of it):

a. Exploratory Testing_____
b. Requirement Testing_____
c. Functional Testing_____
d. Use-Case Testing_____
e. Normal testing_____
f. Positive testing_____
g. Negative Testing_____
h. Stress Testing_____
i. Equivalence Partitioning (or Equivalence classes) ___
j. Boundary Value Analysis_____
k. Random Input_____
l. Random Testing_____
m. Fast Anti-Random Testing_____
n. State Transition Technique_____
o. Model-based Testing_____
p. Mutation Testing_____
q. Fault Injection_____
r. Error-Guessing_____
s. Call-Pair Testing_____
t. Pair-wise Testing_____
u. Call-Tree Method/(Editor) (CTM/CTE) _____
v. Combinatory input testing _____
w. Statement Coverage Testing_____
x. All-definitions testing_____
y. All-Predicates Testing_____
z. All-Computations Testing_____
aa. Permutation Testing_____
bb. Cause Effect Graphing_____
cc. Magic Values_____

17. Which of the above techniques (just give letter(s)) do you feel comfortable with, and use regularly in your testing?_____

18. Which of the above techniques (just give the letter(s)) do you know, but **do not** use regularly in your testing?_____

19. How much to you estimate your everyday testing is dependent on the input values (data) you give? (Much/Medium/Not so much) Please elaborate:
    _____

20. Do you think it would be easy to do many more test cases based on the same one – with small variations, if you just have more time? (Yes, maybe, no) Please elaborate:
    _____
    _____

ID____

21. Now when you have had a walk-through of some of these techniques, do you feel your interpretation of them has changed? (circle them) Then add a value (1 Much, 5 little) of how much you think your interpretation has changed?

    a. Exploratory Testing_____
    b. Requirement Testing_____
    c. Functional Testing_____
    d. Use-Case Testing_____
    e. Normal testing_____
    f. Negative Testing_____
    g. Stress Testing_____
    h. Equivalence Partitioning (or Equivalence classes) ___
    i. Boundary Value Analysis_____
    j. Random Input_____
    k. Random Testing_____
    l. Fast Anti-Random Testing_____
    m. State Transition Technique_____
    n. Model-based Testing_____
    o. Mutation Testing_____
    p. Fault Injection_____
    q. Error-Guessing_____
    r. Call-Pair Testing_____
    s. Pair-Wise testing_____
    t. Call-Tree Method/(Editor) (CTM/CTE) _____
    u. Combinatory input testing _____
    v. Statement Coverage Testing_____
    w. All-definitions testing_____
    x. All-Predicates Testing_____
    y. All-Computations Testing_____
    z. Permutation Testing_____
    aa. Cause Effect Graphing_____
    bb. Magic Values_____

22. Do you consider the walk-through of the test design technique was valuable to you? (Much, Somewhat, Maybe, Nothing new/not at all) _____

ID____

23. Do you wish more time will be spend on discussing these (and other techniques in the future?)

24. Will you CHANGE your testing as a result to this lecture? (Circle)Yes, somewhat, No, Do not know

25. Now: Open the system by double-clicking on the application: You will be asked a couple of questions on the system, and do not download a new version (for this experiment, please). Just say no or cancel….Play with the system to get to know. Please note **how much time you need** to spend to learn the system (you do not need to know ALL aspects of it – but enough to be able to write test cases) _____

26. Test Design Technique (TDT).
    For each TDT create a test case and using the actual technique as closely you can, try to create (write down) one test case that demonstrates the technique that you can immediately execute. Fill in the table next page with times (in minutes) and your opinions.

27. For each Test case use YOUR ID and the TDT nr (so we know what TDT you did). You can also write it by hand on a loose paper (please provide the same Info and an ID).

ID____

NR	TDT	Time to identify Unique part of the system where to apply TC	Grade difficulty to identify where to apply 1= easy 5= impossible	Time to create a unique Test Case (writing it down)	Grade difficulty of creating a Test Case for this technique 1= easy 5= impossible	Estimate (or count) Possible TC you locate for selected target	Do you think this TDT could be used for Your testing?	Comment
1	Normal							
2	Negative							
3	Equival. Partition Eq. Class							
4	BVA							
5	State-Transition							
6	Use-Case							
7	Magic Values							
8	Error Guessing							
9	Permu-tation							

Table edited in size for Thesis.

Appendix 4 Process used for Test Design Technique Evaluation

Process description, experiment for test technique evaluation.

A. Some introduction to the system, context, use, where to find info, how to compile etc "normal" documentation of code, system etc according to Ericsson. Degree and Master Thesis, Programming skills – test interest, Mentoring. Time (4) 6-12-months
B. Chose one or one group of test design techniques (start with on, then add on!)
    1. Input: EqP, BVA, "magic" (Random input)
    2. State-chart/FSM/Model/Cause effect
    3. Mutation (e.g. order….emphasis on)
        a. missing, superfluous
        b. assign (pointers and dynamic bindings)
        c. resource (mem, time, buf, prio)
        d. negative tests (provoke failures, scenarious, fault handling – path, other)
        e. other mutations (tbd) (see published lists)
    4. Coverage (Statement, branch, condition, MCDC)
    5. Search (genetic, hill-climbing, other)
    6. Exploratory (usage-oriented)
    7. Program slicing(?)
    8. Other test design techniques (TBD)
C. Note - All phases below (D, E, F, G, ..) measured for time (in many aspects)& perceived difficulty Evaluate difficulty of learning technique (scale) 1-5 where 5 is difficult (see [63])
D. Understand/define/research Test Technique (write intro paper)
    a. Create examples
    a. Define limitations of technique
    b. Look at definitions and variants (can the technique be expressed "formally" or in an abstract way)
    c. Brainstorm if there are "ways" to automate the technique – document See different "steps" of application

E. Chosen part of systems (within CLS, OSA in CPP?) according to plan. Some overlap (of data)
    a. Explore existing descriptions, specifications, documentation etc
    b. Define/describe/ component/ file(s) where to do experiment (here OUT – object under test) which will also include where it is used, such as interfaces, and "distinct" higher levels (perspective)
    c. Explore current/existing test cases that executes this code…(just as a learning experience) to get perspective (could be omitted)
    d. "Classify" OUT in template provided
F. Create Test cases for OUT– by using the test technique STRICTLY (meaning, no other way to create TC)
    a. Measure time to create each TC (First and second TC etc
    b. Measure coverage for each TC added (could be done afterwards)
    c. How many TC possible to create (if upper limit)
    d. How many TC difficult vs. easy (to create)
    e. Write "evaluation" of test technique (strength – weakness)
    f. During this process, if any hurdles, insights or other clear "faults" are encountered; this must be documented "logged".
    g. Describe "applicability" of technique. Ease of manual testing
G. Automation preparation
    a. Describe approach to automate TC (implementation proposal)
    b. Describe architecture of automation
    c. Describe your assumption of ease of automation (time to)
H. Automate suite of TC (if possible and relatively easy)
    a. Document suite
    b. Measure time to automate (1st, 2nd TC etc)
    c. Are there "similarities, patterns" in the automation? Describe in text
    d. Independent evaluation of automation (code)…

I. For each TC, now create a “real” software fault that will make each TC fail
    a. Document time to create fault
    b. Document each fault (more than one “type” of fault for each TC is appreciated)
    c. Measure Coverage (and changes)
J. Repeat -I on “next level” of OUT as defined above in Eb.
K. For each fault add “search-replace” and describe in fault injection system how that would happen
L. Use fault injectors (i.e. Use “mutations “and faults” from others) and evaluate which TC holds and not with new set of faults
    a. Then it will be a matter of checking if the new faults
        i. can be moved (how many will be evaluated)
        ii. Many faults at the same time (or combination)
        iii. If they are visible at higher levels (after J)
        iv. And actually will make any test cases fail (and coverage)
M. Check if your “automation of technique” could be transferred to other code (and what it would need to do so) Document how
    a. If possible (and time left), try automation on other OUT