2011 Fourth IEEE International Conference on Software Testing, Verification and Validation

Analysis of Mistakes as a Method to Improve Test Case Design

Sigrid Eldh[1,2]
[1] Radio System and Technology
Ericsson AB
Stockholm, Sweden
Sigrid.Eldh@ericsson.com

Hans Hansson[2]
[2]School of Innovation, Design, & Eng.
Mälardalen University
Västerås, Sweden
Hans.Hansson@mdh.se

Sasikumar Punnekkat[2]
[2]School of Innovation, Design, & Eng.
Mälardalen University
Västerås, Sweden
Sasikumar.Punnekkat@mdh.se

Abstract— **Test Design – how test specifications and test cases are created – inherently determines the success of testing. However, test design techniques 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 test design techniques, leading to an improved coverage without loss of efficiency.

Keywords: **Test Design, Test Case, Improvement Method, Test Techniques, Efficient Testing**

I. INTRODUCTION

In academia, a test design technique 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 test design techniques, 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.

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.

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 [7], [9].

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. This paper is structured as follows: First we provide an overview of test design, including terminology and related work. Then we present the study from where we have collected the data. Section IV contains a detailed description of our categories of mistakes. In Section V, we compare our categories with industrial test cases. Section VI presents our proposal for improving test case design in practice. Finally, we conclude with threat analysis, discussion and further work.

II. 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 [1] (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

978-0-7695-4342-0/11 $26.00 © 2011 IEEE
DOI 10.1109/ICST.2011.52

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 [6][8]. Improvement models as a general modus operandi is found in e.g. [3]. Particular work on improving the test design phase and assessing the test cases in this manner is, to our knowledge, new.

III. 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.

A. *Process of this Study*

The process of data collection for this study is described in Figure 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 test design techniques and their usage.

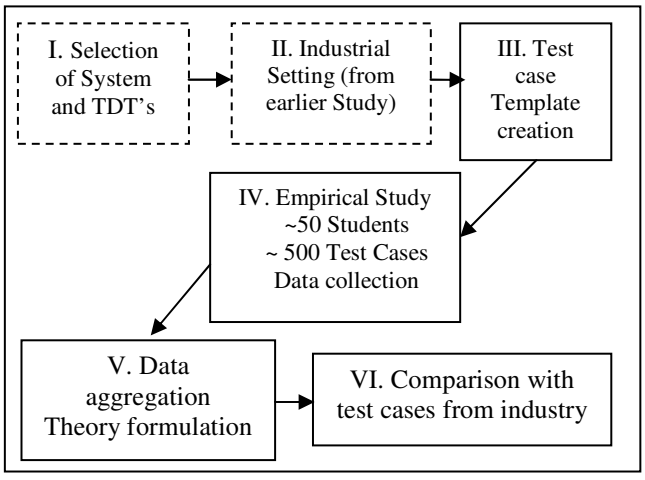

Figure 1. Process of the Study

A simple open source system, Buddy [5] 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 paper (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 test design techniques, or if it was due to a lack of know-how of the test design techniques, 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 test techniques 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 1).

B. *Empirical Study and Data Collection*

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, Buddy [5], 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 test techniques. Each student was asked to use a series of test techniques 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.

C. Test Case Template

In this study we used a test case template based on IEEE Standard 829 [1] 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 [10]. 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).

D. Considered Test Design Techniques

In this study, the following test design techniques were explicitly taught in theory, including some simple examples:

- The positive test case (valid input data) (Pos T) [15], [12], [13], [21]
- The negative test case (invalid input data) performed twice (Neg1 + 2) [14], [17], [19], [21]
- Magic input test case (0, -, float or other typical fault invoking data) [4]
- Equivalence partitioning technique (EP), [11] also referred to as Category Partitioning
- Boundary Value Analysis (BVA) [4], [11], [15]
- State-transition – preparing the model for the test case (based on the system) (STModel) [20]
- 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!) [6]
- Combination techniques: State-transition + input analysis (Add EP-classes) (Comb)

IV. Systematic Mistakes Analysis

The data collected in the above study was aggregated, and treated statistically, as indicated in Figure 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 test design techniques 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.

A. *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.

B. *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.

C. *Understanding Test Design Techniques and How to Apply Them*

One can discuss the test design techniques 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 test design techniques 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 test design techniques 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 test design techniques by the students is shown in *Figure 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!). 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.

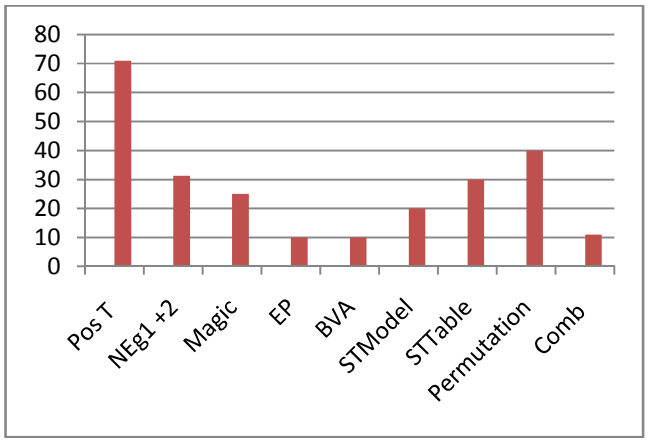

Figure 2. Test Design Techniques % Full and partial success in constructing executable test cases in the technique.

When using test design techniques to create test cases, the first aim should be to create test cases to gain as much coverage [21] 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.

D. 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.

E. 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.

F. 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.

G. Specification of Valid and Invalid Inputs

Defining input classes in the test/verification specification simplifies the use of different test design techniques, 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 test design techniques. 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 test design techniques 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.

H. 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.

I. 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.

J. 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.

V. 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 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**)

A. 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 test design techniques 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.

B. 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.

C. 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 test design techniques, not specifying input ranges etc. in specifications.

D. 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 [18] has been used, which requires a clarification of the valid and the invalid by defining the borders (min – max). This is a good step forward for improving the test coverage.

E. 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.

I. 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 3*.

This process can be started at two steps, either by assuming the categories in this paper as an initial start – or by using the proposed test case template which would be at phase e in *Figure 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 test design techniques (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.

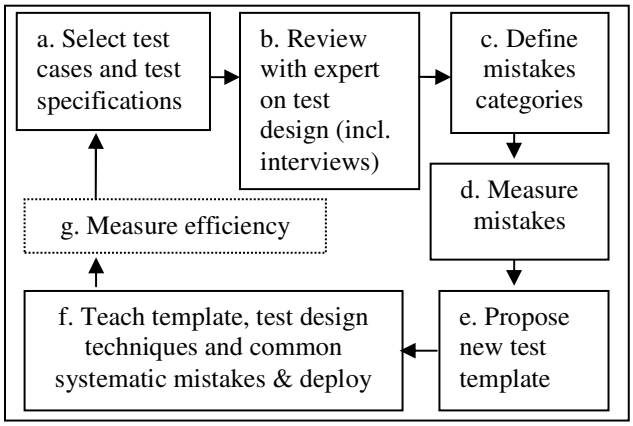

Figure 3. Systematic Mistake Elimination Method

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.

II. THREATS TO VALIDATY

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 test design techniques in *Figure 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.

III. Conclusion And Further Work

This paper 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.

Acknowledgment

We would like to thank Ericsson AB for funding our work and for allowing us to publish these results. The Knowledge Foundation is acknowledged for funding this work through the SAVE-IT program.

References

[1] IEEE Std. for Software Test Documentation 829-1998 & 2008

[2] Basili, V. and Elbaum, S. 2006. “Empirically driven SE research: state of the art and required maturity”, Int. Conf Soft. Eng., ICSE, 2006

[3] Basili, VR. Rombach, HD., “The TAME project: Towards improvement-oriented software environments”, Trans. of Softw. Eng. June Vol 6. 1988

[4] Beizer, B. *Software Testing Techniques,* Int. Thomson Computer Press, 2nd ed., Boston, 1990

[5] Buddy system http://buddi.digitalcave.ca/index.jsp

[6] 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

[7] Eldh, S., “On Evaluating Test Techniques In An Industrial Setting”, Mälardalen Uni. Lic. Thesis No.78, 2007

[8] Eldh, S., Punnekkat, S., Hansson, H., Jönsson., P.: Component Testing is Not Enough - A Study of Software Faults in Telecom Middleware, *Proc. 19th IFIP Int Conf. on Testing of Comm. Syst TESTCOM/FATES*, Springer , LNCS, Tallinn, Estonia 2007

[9] 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.

[10] Test Case Template & Additional data and information, regarding Systematic Mistakes in Test Design Study: http://www.idt.mdh.se/~seh01/ICST2011/

[11] Jorgensen, P. “*Software Testing: A Craftsman’s Approach*”, Department of Computer Science and Information Systems, Grand State University, Allendale, CRC Press, 1995

[12] King, J. C. A new approach to program testing. In *Proc- of the Int. Conf. on Reliable Software* (Los Angeles, California, April 21 - 23, 1975). ACM, New York, NY, 228-233. 1975.

[13] King, J. C. x Symbolic execution and program testing. *Commun. ACM* 19, 7 (Jul. 1976), 385-394. 1975.

[14] Legeard, B., Peureux, F., Utting M., Automated boundary testing from Z and B, Lecture Notes in Computer Science, Springer Verlag, 2002

[15] Myers, G. *The Art of Software Testing*. John Wiley & Sons Inc, USA, 1979

[16] Murnane, T., Reed, K., Hall, R., “Tailoring Black-box methods”, ASWEC , 2006

[17] Nguyen, D. C., Perini, A., and Tonella, P., A Goal-Oriented Software Testing Methodology, V. 4951, p. 58-72, LCNS, Springer Verlag, 2008

[18] QuickCheck http://www.quviq.com/

[19] Whittaker, J. A., How to Break Software: A Practical Guide to Testing, Addison-Wesley, 2003

[20] Whittaker, J. A., Thomason, M.G., “*A Markov Chain Model for statistical software testing.”* Transactions on Software Engineering, VOL. 20, No. IO, Oct 1994

[21] 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.

[22] Zhu, H., Hall, P.A.V., May, J.H.R., “ Software Unit Test Coverage and Adequacy”, ACM Com. Surveys, Vol. 29, No.4 Dec 1997.