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).

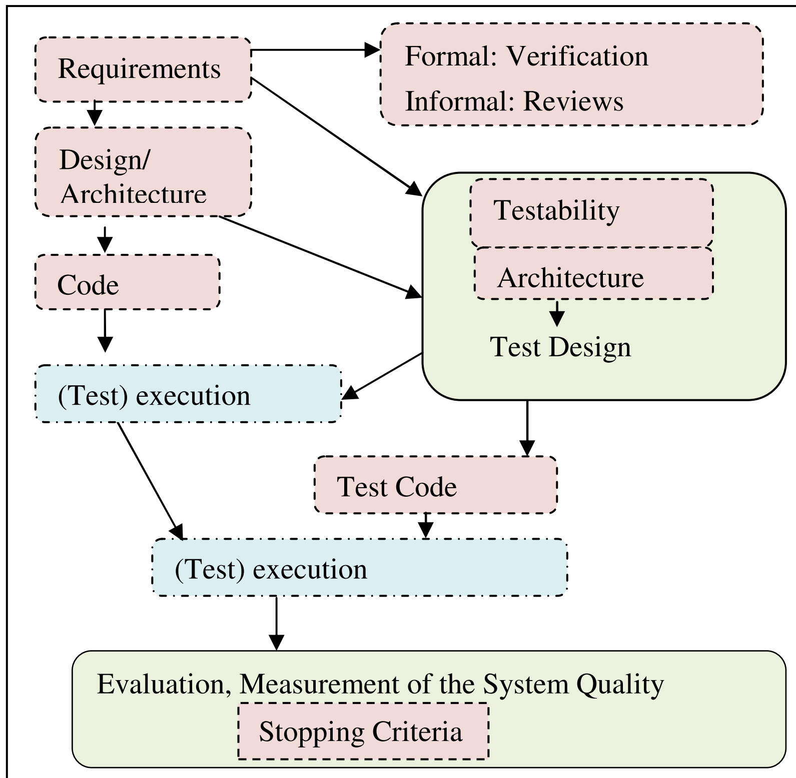

Figure 2.1 Overview of targeted areas in this thesis.

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.

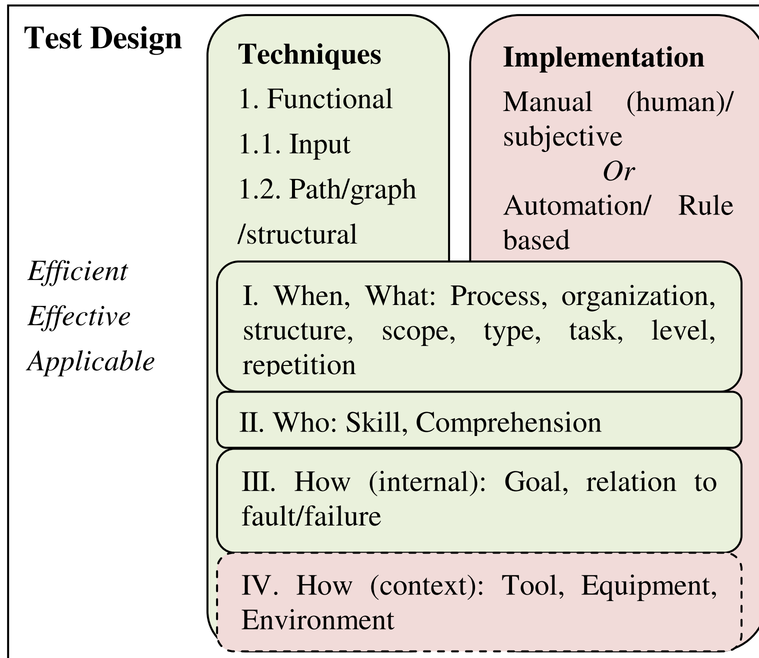

Figure 2.2 Targeted areas of Test Design: Another perspective

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).

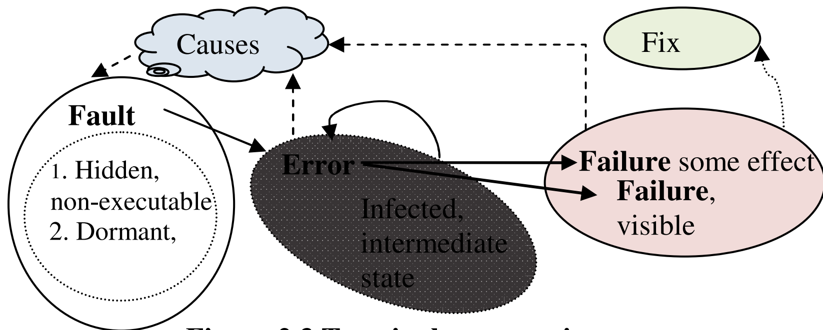

Figure 2.3 Terminology mapping

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?”

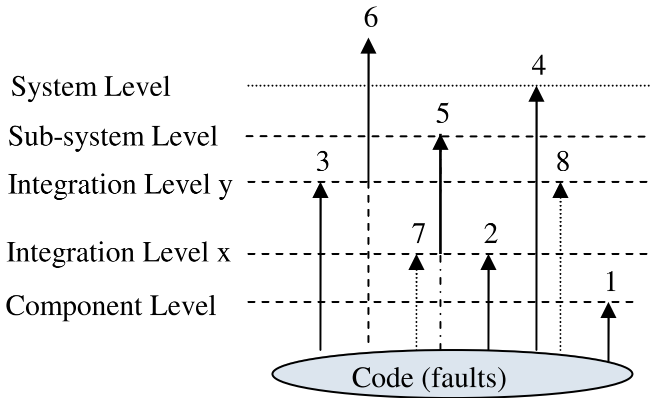

Figure 2.4 Fault propagations to failures, can be captured at different levels

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

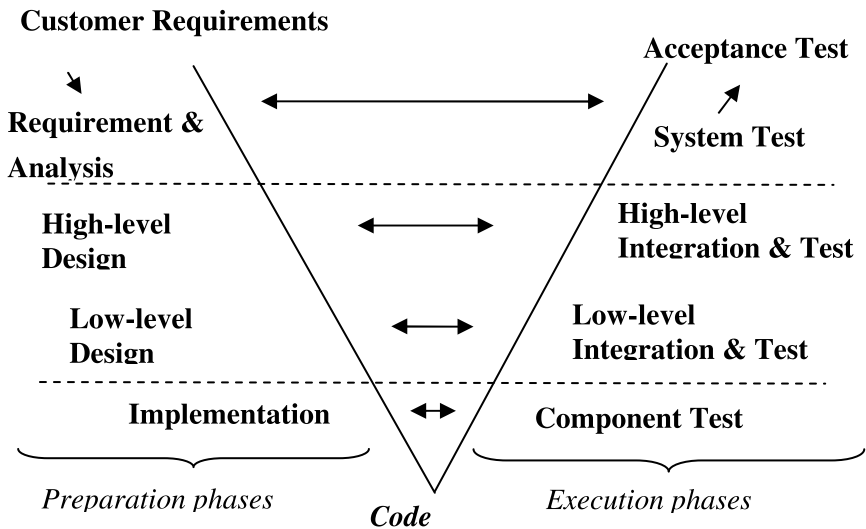

Figure 2.6 V-model – where preparation and execution phases for test are shown

[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

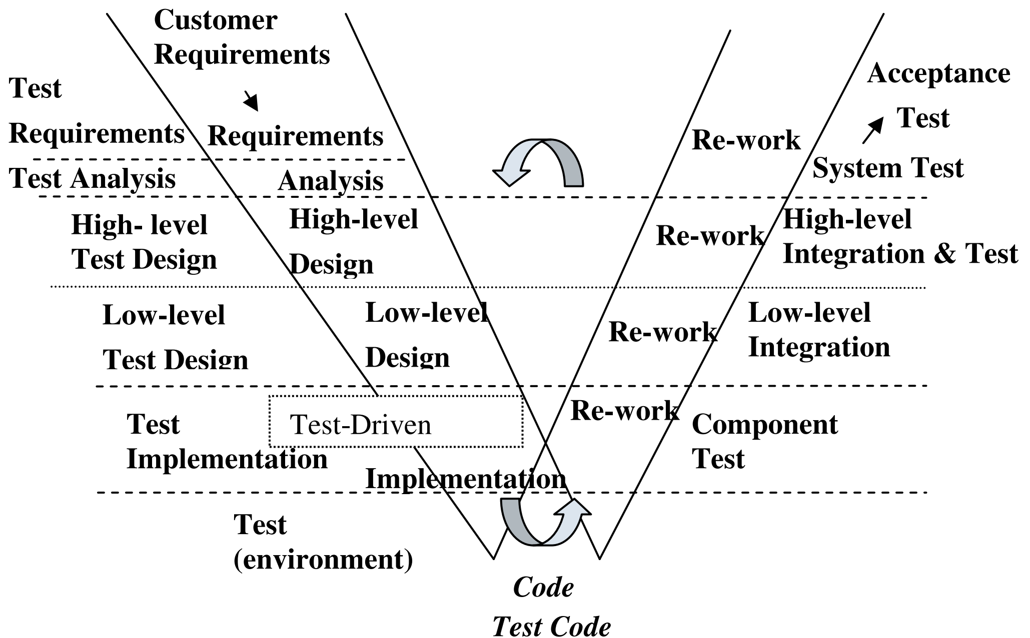

Figure 2.7 The W-model adapted to industrial parallel & iterative/incremental design

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.