Master Thesis in
Software Engineering
Thesis no: MSE-2001-14
October 2001

Test Process Evaluation by Combining ODC and Test Technique Effectiveness

Dan Bengtsson

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE - 372 25 Ronneby
Sweden

This thesis is submitted to the Department of Software Engineering and Computer Science at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:

Author:
Dan Bengtsson
Address: Västra Stationstorget 7, 222 37 Lund, Sweden.
E-mail: danbengtsson@hotmail.com

External advisor:
Tord Kroon
Ericsson Software Technology AB
Address: Ölandsgatan 1-3, 371 23 Karlskrona
Phone: +46 (0)455-39 50 00

University advisor:
Claes Wohlin
Department of Software Engineering and Computer Science

Department of
Software Engineering and Computer Science
Blekinge Institute of Technology
Box 520
SE - 372 25 Ronneby
Sweden

Internet: www.ipd.bth.se
Phone: +46 (0)457 38 50 00
Fax: +46 (0)457 271 25

Abstract

This report discusses the importance of test process evaluation in order to improve a test model and to provide developer- and management feedback. The report results in a test evaluation framework, developed in cooperation with a department at Ericsson Software Technology in Karlsrona. The framework is a result of discussions with the developers regarding performed testing, studying defect types from past projects and by analyzing the result from a small survey answered by some of the developers at Ericsson.

The overall project aim was to evaluate performed testing in order to improve the test model. This requires a good insight of the test process, which is provided by the developed test evaluation framework. The test process is visualized by extracting test process data, making it possible to achieve the project aim. The project aim can be divided into the three following areas: *Firstly* to evaluate if the current test model is followed as expected, for example are all test techniques used according to the test model? *Secondly* to evaluate how well the test model fulfills predefined expectations, i.e. is a defect detected with the expected test technique and in the expected test phase? *Finally* to evaluate if there are any problematic defects that should receive extra attention during a project such as if one or several defect types occurs more frequently than others?

The framework is based on another framework, Orthogonal Defect Classification [Chillarege92], combined with the research area Test Technique Effectiveness. The aim of this combination was to support the developed framework. Further a specific part of the framework is focusing on developer- and management feedback.

Key words: ODC, Test Technique Effectiveness, Test process evaluation, Developer- and management feedback.

1.0 Introduction

When developing software it is essential to build reliable products, which in this report is defined as a product that fulfills the system requirements and with a low number of defects. The project aim was to develop a test evaluation framework that can evaluate:1) How well the current test model is followed, for example are all test techniques used according to the test model? 2) How well does testing fulfill predefined expectations, i.e. is a defect detected with the expected test technique and in the expected test phase? 3) Are there any problematic defects that should receive extra attention during a project such as, if one or several defect types occurs more frequently than others?

The project aim is closely related to the test process and product reliability, since this is where all the defects should be detected. However the focus in the report is not on improving the test process, but to provide information that can be used in order to improve a test model. Good understanding of the test process is necessary for this, otherwise it is difficult to know how improvements can be made. To gain better understanding of the test process, data has to be extracted from projects and analyzed. In the developed framework this is done by logging occurred defects and performed testing, which then can be used for evaluation of the test process and provide decision material for test process improvement.

Another area discussed in the report, that is intrinsically liked to the evaluation of the test process, is defect classification. A poorly defined defect classification results in that no correct evaluation can be made of the test process, since the classification can lead to misunderstandings of how to classify a defect. Conclusions based on such classification may result in wrong process improvement decisions.

The project is a result of the need of a test model and increased test awareness at the department U/PD, at Ericsson Software Technology in Karlskrona. The stated requirements aim to increase the software reliability by 1) identifying possible factors that can improve the test awareness of the developers, and 2) to develop a test model for the developers of *what* and *how* to test.

After studying the test process and defect reports at U/PD, a combination of research areas. The identified areas are: *Fault Classification* and *Test Technique Effectiveness*, and the defect classification model *Orthogonal Defect Classification*.

- Fault Classification is a classification of fault types that can occur in a product. Fault Classification is the base for visualizing and improving the test process [EmWi98]. Without such a classification it is hard to make any accurate statements about the software reliability. Fault Classification can also be defined as Defect Classification, which is the term used in the rest of this report.
- Test Technique Effectiveness (TTE) is based on the theory that test techniques are not equally good at 1) detecting defects or 2) detecting defects of a specific type. An increased understanding for this area, would mean a better knowledge about which defects that may remain in the product after performed tests.

- The Orthogonal Defect Classification (ODC) is a defect classification model, providing a framework to 1) identify development problems related to project phases, 2) evaluating performed testing compared to expectation on test techniques and test phases, and 3) the identification of problematic defect types [Chillarege92].

Even though the developed framework in this report has several advantages, there are some problem areas related to the research areas previously described. *Defect Classification* suffers from difficulties of constructing a classification that is 1) simple enough to understand in order for two developers to classify a defect under the same category, and 2) include all possible defect types related to a product. The other problem area concerns *Test Technique Effectiveness* that has proven to be a difficult area. The main reason for this is that the result of testing depend on the development environment in which the experiments are performed such as: program type, developer experience, programming language etc. This results in that performed experiments are difficult to compare and in some cases conclusions are contradicting.

Report structure:

- The following chapters describe the background of the project (*Chapter 2.0*), project hypotheses (*Chapter 3.0*) and method used to verify the project hypotheses (*Chapter 4.0*).
- In order to understand the terminology used in the report, the next part of the report provides basic knowledge about testing such as test phases and test techniques *(Chapters 5.0 - 6.0)*.
- *Chapter 7.0* covers the research area TTE and *Chapter 8.0* describes the ODC framework.
- Since the developed framework should be adapted to the development- and product environment at U/PD, a description of the defect classification and test techniques used at U/PD is given in *Chapter 9.0*.
- After covering the areas used in the developed framework, ODC, TTE and the development- and product environment at U/PD, the suggested framework is described in *Chapter 10.0*.
- *Chapter 11.0* concludes the report and validates the hypotheses, *Chapter 12.0* suggests future work and -research, and *Chapter 13.0* contains the report references.
- Finally *Chapter 14.0* is an appendix and contains a description and summary of the performed survey, and also discusses the conclusions drawn from it.

2.0 Project Background

The reason for the project is that U/PD identified the need for a test model that could improve the developers' test awareness in general. The test model should be adapted to the development environment for the Service Data Point (SDP) product developed at U/PD, which is situated within the Ericsson Product Unit "Charging Solutions". The department is situated in Karlskrona and consists of 30 programmers. The task for the department is to develop the SDP product that is the heart in the Ericsson PrePaid solution (PPS). The SDP is a distributed real time system with telecom characteristics, i.e. high availability. The SDP is built upon SUN standard products and its main functions are rating of calls and account handling for PrePaid subscribers.

The testing of the SDP is performed on several levels, which reduces the number of defects in the end product. However the longer a defect remains in the product the more it costs. Therefore this report is focusing on the testing performed by the programmers, since this is one area where the detection of defects can be made early. The type of testing performed by the developers is a decision that each programmers has to make, which can result in various test results between the developers. Currently there is 1) no logging of what is tested and 2) no well defined defect classification making it hard to visualize the test results or to identify possible improvement areas. These two areas were identified as the main areas where improvement would lead to a good test model and increased test awareness for U/PD.

3.0 Hypotheses

The general idea with this project is to extract data from projects for evaluating the test process, and also to provide developer and manager with feedback on performed testing in relation to occurred defects. The statements below are designed to achieve the project aim.

1. There has to be a classification of defect types that is well understood, i.e. the risk that two developers classifies a defect under different categories has to be minimal.
2. The defect classification should optimally represent all possible defect types that can occur in a specific environment.
3. In order to be able to analyze how well the current test process works, defects and testing activities has to be logged.
4. The defect types have to be mapped towards one or several test techniques to identify test process problems easier. For example if an occurred defect is the result of that a test technique has not been used.
5. There has to be defined expectations on which defects that is expected to be detected by a certain test technique and in which test phase.
6. Developers and managers have to receive test process feedback of which defects are made in relation to occurred defects, in order to prevented them in the future.

All the requirements are intrinsically related and necessary in order to achieve the project aim. The first two requirements, regarding defect classification are required in order to perform any process evaluation, which is essential for all the parts of the project aim. The third statement is a direct result of the first two statements in the hypotheses, *firstly* in order to be able to analyze why a certain defects occur they have to be logged in some way, and *secondly* in order to draw correct conclusions based on these, the logging of the defects has to be correct.

The next two requirements (4-5) are related to the second and third part of the project aim, i.e. evaluate the expectations on the test techniques and test phases. They are also necessary in order to provide developer- and management feedback, which is the last statement in the hypotheses. The final statement aims to fulfill the first project aim, to evaluate how well the testing is performed. For example if a developer has not used all the test techniques according to the test model he/she should be notified of this. Further the last statement relates to the third project aim of identifying specifically problematic defect types, since this information should be provided both to developers and managers.

4.0 Method

This project evolved from the requirements identified at U/PD, to develop a test model that could increase the test awareness at U/PD. These requirements lead to three phases:

1. To gain understanding of how testing is performed at U/PD by studying occurred defects in past projects and performed testing.
2. To develop a test model supported by research.
3. To develop a test evaluation framework based on ODC and TTE.

The *first* part was achieved by discussing how each developer performed test, and by handing out a minor questionnaire, and studying defect reports. The *second* part was planned to be based on the information retrieved from the developers and the questionnaire, and also by identifying different research areas that could validate the test model. The insight gained from the two first phases lead in a third phase that resulted in the final project aim and hypotheses.

4.1 Phase one

The first phase was characterized by informal discussions with the developers, studying defect reports and developing a questionnaire, in order to understand the current test process. The questionnaire focused on four areas listed below. The conclusions drawn from the survey is described in *Chapter 14.0*.

- *Defect types:* Which type of defects that was the most occurring ones, hardest to locate and to correct?
- *Test Cases:* How and when test cases are designed?
- *Tools:* To what extent available testing tools are used?
- *Inspections:* Which types of defects that the developers found with inspections and which attitude they had towards the technique?

The reason for choosing these areas was in hope to identify relations between occurred defects and performed testing. For example if badly performed testing or the lack of using a certain test technique was the source of occurred defects. However because of the defect categories used at U/PD are ambiguous and developed for another product than SDP, it was difficult to draw any conclusions based on these. Since each defect is described in detail it would be possible to categorize them by using another defect classification. However this would be a time consuming task and since there is no logging of which tests that has been performed, it would have been very difficult to find any relations between occurred defects and performed tests. This lead to the second phase of the project, trying to develop a test model that could be validates from research result.

4.2 Phase two

The second phase of the project consisted of studying the research area TTE and Defect Classification, which were the areas believed to provide support to the developed test model. Unfortunately it became clear that experiments performed on TTE were performed under different circumstances and the conclusions were in some cases contradicting. Consequently it was difficult to develop a test model that could be validated by research results, which lead to the third phase of the project.

4.3 Phase three

The insight gained from the first project phases: 1) performed testing at U/PD could not be evaluated, due to that the current defect classification was ambiguous and developed for another product, and 2) that a test model could not be validated by research results, which lead to the project aim of developing a test evaluation framework that provides project data for possible process improvements.

During the project we found that the ODC framework fitted the project aim and is therefore the main area of the report. The idea was to develop a test evaluation model based on ODC, that could evaluate the current test model, but on a more detailed level. Further there should be a focus on developer feedback in order to achieve U/PD's requirements of increased test awareness. The developed framework does not consist of a test model that was one of U/PD's requirement, but can be developed and improved based on the process feedback provided by the framework.

5.0 Testing Fundamentals and Project Terminology

In order to understand the terminology used in this report and how they relate to the discussed areas, three questions are stated: *why-*, *what-* and *how to test*? The answer to the first question, is to verify software reliability (*Section 5.1*) and is one of the best ways to assure that a product fulfill the system requirements. The question *how (Section 5.3)* to test depends on *what* (*Section 5.2*) that should be tested and also on several parameters such as: 1) the development environment, 2) system functionality and 3) possible defect types for a specific system. *Figure 1* illustrates the relation between the areas.

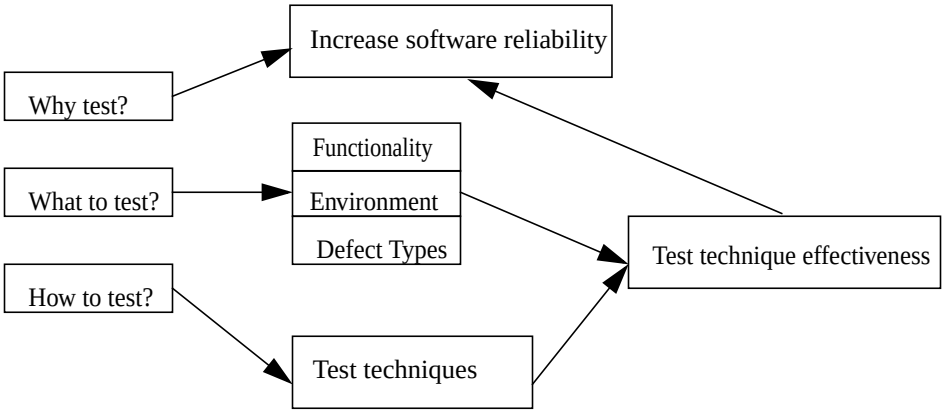


FIGURE 1. The figure visualize the relation between the three stated questions: why, what and how to test.

5.1 Software reliability

This section describes the importance of testing and how it can provide a measure of the current software reliability. Software reliability can also be used for predictions of how the software reliability is likely to evolve during a project, which then is useful for the estimation of project time.

Software reliability is a measure of the likelihood for a program to execute in a specific environment without the occurrence of failures and with the respect of time [Wohlin]. The difference between failure, defect and a fault is described in *Section 5.2.2*. In order to measure current software reliability, defects have to be detected and logged. The detection is done by the use of different test techniques, which makes testing an important factor aiming for improved software reliability, i.e. the more defects that are detected the higher chance for increased software reliability.

Previously software reliability was defined as: a low number of defects in a product. However this would mean that a program with 100 lines of code (LOC) containing 5 faults, has better quality than a program with 50 000 LOC containing 10 faults, which most people would agree is wrong. Therefore a better definition of software reliability is the measure of 1) *Mean Time Between Failure (MTBF)* or *2) failure rate*. The MTBF is the time between the occurrence of two failures *(Figure 2)* and the *failure-rate* is referring to the frequency of failures during the execution of a program [Wohlin].

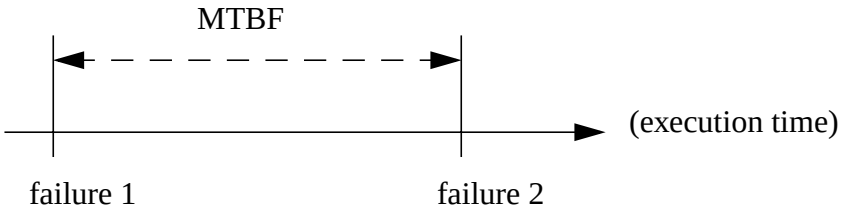


FIGURE 2. The Mean Time Between Failures is a measure of the average time between failures.

5.1.1 Software reliability growth models (SRGM)

An important issue of software development is the estimation of project time. This makes the reliability aspect very important, since low reliability can affect the development time needed to reach expected quality, and consequently plays a vital role for the delivery date. The ODC framework, described in *Chapter 8.0*, can be used together with SRGM, which is the reason for describing the technique here.

In order for to make statements about the software reliability, we believe that a project manager has to be able to answer two questions during a project:

1. Which is the current software reliability?
2. How much more development time is needed in order to deliver the product with required reliability?

These two questions can be answered with the help of SRGM. They can verify and predict the reliability, and indicate potential problems that otherwise could jeopardize the planned delivery date. A project manager can use this type of information as decision material for allocation of more programmers or to decide if the customer should be notified of a delayed delivery.

SRGM consists of two parts in order to measure and predict reliability:

- Defects are collected and inserted in a function with respect to time and number of defects
- A mathematical function that is fitted to the defects and show the future defect-detection rate.

Figure 3 gives a brief overview of how the prediction of software reliability is done: 1) the first picture in *Figure 3* shows the number of occurred defects in a project at a given time, 2) the second picture shows a mathematical function based on defect history, and 3) the final picture shows when the mathematical function is fitted to the current defect-detection rate.

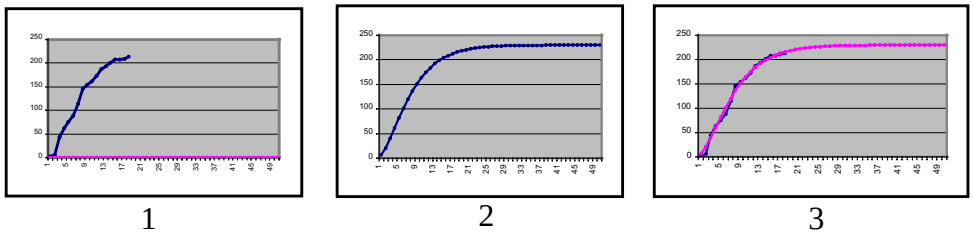


FIGURE 3. The figure shows how the number of defects detected in a project is fitted to a mathematical function, which makes it possible to predict the failure-rate.

5.2 What should be tested?

Testing is crucial for the assurance of software reliability, however it is difficult to know what to test. The decision depends on the development environment and is therefore different between projects. We have recognized three areas of importance when determining *what* to test:

1. *System functionality*: The functionality of a system is probably the most obvious area when testing. Both functional- and non-functional testing should be considered.
2. *Development- and product environment:* The development- and product environment is affecting the way a program should be tested. For instance using the programming language C++, the test process should include test defect types related to memory handling, i.e. memory leaks, allocation management etc.
3. *Defect types:* The defects that can occur in a system depend on the development- and product environment.

Further the decision of what to test, depend on which phase the project has reached, since different types of testing are performed in different phases. Software is built in portions, smaller units are developed and tested. These are then integrated to a somewhat larger unit and tested again, i.e. integration test. Each phase is responsible for testing different aspects of the product (*Section 5.2.1*). The next two sections briefly describes these phases and their responsibility for detecting different defect types (*Section 5.2.2*).

5.2.1 Test phases

A project is usually divided into four major phases: Analysis, Design, Implementation and Test. Even though all these phases can provide value to software reliability, the test phase is the only stage in a project where software reliability can be verified. The test phase is divided into different phases, aiming to test different aspects of a system. Figure 4 shows the relation between the test phases.

FIGURE 4. The figure shows the relation between the different test phases.

1. *Unit Test (UT):* is usually performed on a minor part of the code, performed by the author of the code and aims to detect defects that otherwise results in a failure when executing the program. This is also the phase where units are integrated to larger units and tested again, so called integration test. This type of testing can also be performed in the test phase described next, i.e. Function Test.
2. *Function Test (FT):* is performed by a tester and specifically tests the product functionality, independent of code structure. This is done by analyzing input and corresponding output. Wrong output means that a failure has occurred. Optimally no code related failures should be detected on this level, but rather failures related to design.
3. *System Test (ST):* is similar to function test, but is performed in an environment similar to the customers' target environment. The type of system defects that should be detected on this testing level are those related to the system requirements, for example performance.

There are general rules of what should be expected from each test phase, however it is common that defects are detected in other phases than expected. For example if it is decided that unit test should detect defects of the type *memory leaks*, they should not be detected in Function or System Test. This relation is visualized in *Figure 5*, where each test phase has a relation to a corresponding phase in the development process, i.e. Analysis, Design and Implementation.

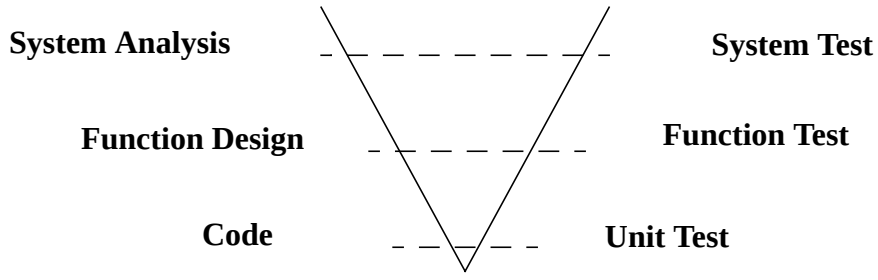


FIGURE 5. The relation between test phases and other phases in the development process.

5.2.2 Error, fault and failure

Previously the incorrectness of a program has been referred to as *defects* and *faults*. Besides these definitions, *error* and *failure* are two other terms used to describe incorrectness in a product. The differences between these is defined below [IEEE90]:

1. **Error:** A mistake that produces an incorrect result. For instance, a mistake can originate from a misunderstanding of the requirements specification and can lead to both a *defect* and *failure*.
2. **Fault and Defect:** An incorrectness in a program caused by a mistake. A fault may cause a failure if being executed.
3. **Failure:** Occurs when the behavior during the execution of a program differs from the behavior described in the requirements specification for a program.

5.3 How to test?

When an analysis has been made of what should be tested in a product it remains to identify the best way to perform the tests. There are several different test techniques that can be used when testing, the question is which technique that is best at detecting defects?

Test techniques can be divided into four groups: 1) *Functional-, 2) Structural-, 3) Static testing* and 4) *Testing tools*. These in turn consist of several different test techniques *Chapter 6.0*. Which of the techniques to use and when, has been evaluated in research, showing that different test techniques are not equally good at detecting different defects and failures. The research area is referred to as *TTE (Section 7.0)*, previously defined as a measure of how many defects that a test technique can find in a program. Note that the time aspect is not considered in this definition.

An area that includes the time aspect when detecting defects is *test technique efficiency.* However this measure is not discussed any further in this report. The focus is not on how fast a defect can be found, but rather to identify how a defect can be detected, which leads us to the next chapter describing the different test techniques.

6.0 Test Techniques

Related to what previously was said in about how to test, this chapter describes some of the different test techniques available for testing. The test techniques described are: 1) functional-, 2) structural- and 3) static testing. Functional- and structural testing is based on test case design and have the common factor of that they have to be executed in order to detect defects. Test cases for functional testing have the specific characteristics of only considering program functions without any knowledge of the code structure, where as test case design for structural testing specifically consider the code structure. It should also be noted that structural testing only can be performed in unit test, where as the function test can be performed in both unit- and function test.

Static testing differs from the previous test areas in that neither test case design nor program execution is needed in order to find defects. The type of static testing that is described in this report is referred to as inspection. Inspections can be used for inspecting any type of document, but we focus on code documents.

6.1 Function test (black box)

Function test, sometimes referred to as *black-box testing*, focuses on detecting failures related to program functionality, which is done without considering the code structure. This is characteristic for the Function Testing phase, but this type of testing is also performed during unit test, since the functionality can be tested on different levels.

The test cases for function test are extracted from the requirement- and program design specification, where *input data* and its corresponding *output data* are identified. An example of how this analysis is done is visualized *Figure 6*. Consider the program *square-root*, where a square-root of an input value (x) should result in the returning value (y). If x=4, the output value y, should be 2. Other output values are considered as a failure.

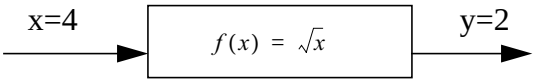


FIGURE 6. A function square-root with the input value= 4 and corresponding output value = 2.

This seems straightforward, but it can be difficult to identify the test cases needed to assure that the risk for that the occurrence of defects is minimal. Further if all values that can be used as input to a program were to be tested, the test process could go on forever. Also in many programs the number of input values are infinite. A better way is to identify input- and output data from the program specification, and divide them into groups with the same characteristics: *equivalence partitioning* (*Section 6.1.1*).

6.1.1 Equivalence

The aim of equivalence partitioning is to reduce the number of test cases for a program and still be confident of that most defects are detected, which is done as mentioned before by dividing *input*- and *output data* into domains. Examples of common domain characteristics can be *positive or negative integers*, *integers* within a specific interval, *prime numbers* etc.

Provided that the domains are correct it should be enough with one test case per input domain, meaning that no matter which value that is chosen from a domain, it should result in the same output domain.

Example 1: A simple example is used to illustrate *equivalence* partitioning. Consider a function that shall identify the gender of a person from their personal id. For a Swedish citizen this number contains 10 digits, where the ninth digit shows if the person is a male (odd numbers) or a female (even numbers). By dividing the input values into two domains, 1) one domain with even numbers and 2) the other with uneven numbers, it is possible to construct a program that can determine a persons gender.

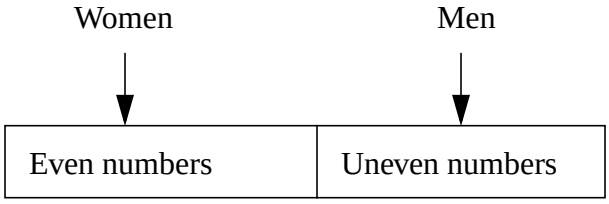


FIGURE 7. The partitioning of numbers between 0 - 9 into two different domains, where even numbers represent women and odd numbers represent men.

When designing test cases for this function, it should be enough with one value from each domain. For example 2 and 3 to represent the interval between 0-9. However from a programmer's perspective more test values should be identified in order to be sure of that the function does not generate any failures during execution. This is best performed by using the *boundary value selection* technique, which is described next.

6.1.2 Boundary

Even though input data to a program has been divided into domains it is important to identify a domain's boundary values as well, since this is where defects often occur. *Figure 8* shows how the boundary values are chosen from the domain identified in the gender example in previous section. Two new partitions have been added in order to identify the boundary values: -1, 0, 9, 10.

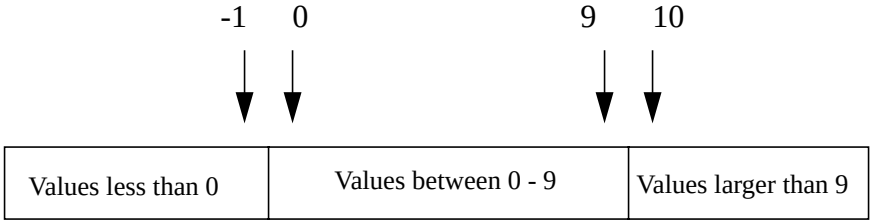


FIGURE 8. The identified domains and boundary values for the gender example from *Section 6.1.1*.

Many test cases can be identified by designing test cases in function test, but not those that are specific for the code structure. This will become clear in the next section.

6.2 Structural testing (white box)

Structural testing, also referred to as *white-box testing*, is a test technique that unlike function testing considers the code structure. The two test areas are complementary. Function testing identifies the main domains and boundaries, extracted from the requirement specification and program design. Structural testing on the other hand is used to identify more domains within the domains found in function test. These can only be identified when analyzing the code and consequently more domain- and boundary values can be identified that otherwise would have been missed. It is often the case that specific requirements have to be solved in a certain way, not originally thought of, which results in more test cases.

6.2.1 Boundary value selection continuing

The importance of code analysis is showed in Example 2. A group of people's personal id is used as input into the function *determineGender1*. Previously it was established that it should be enough with two test cases to determine a gender based on a personal id. When analyzing the code in the example it is evident that it is possible to identify more test cases.

In order to assure that all paths of relevance for even numbers between 0 - 9 works according to the program specification, there has to be one test case for each case-statement. If the value 4 is chosen to represent the domain of even numbers, the other paths in the case-statements are not verified and could contain defects. In the example the case-statement for number 6 is missing, and if a test case containing 6 as an input value is not designed, a failure will occur when the system is going live.

Example 3 shows a better solution to the problem where it is enough with only two test cases. This due to that modulo (%) is used to determine even- and odd numbers. Note that the number of test values identified when performing the code analysis does not determine if the solution is good or not.

Example 2: The function checks the gender of a group of people by analyzing incoming personal id's. The solution demands several test cases due to the use of case-statements.

```
function determineGender1(Array personal_Id_List){
    for(int i=0; i < personal_id_list.length(); i++){
        int digitNine = getDigitNumberNine(personal_Id_List.elementAt(i));
        boolean evenNumber = false;
        switch(digitNine){
            case 0: evenNumber = true;
            case 2: evenNumber = true;
            case 4: evenNumber = true;
            case 8: evenNumber = true;
            if(evenNumber){ print("The person is a female!");}
            else{print("The person is a male");}
        }
    }
}
```

Example 3: The function checks the gender of a group of people by analyzing incoming personal id's. This example only needs two test cases because of a different code structure.

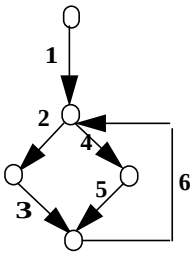
```
function determineGender1(Array personal_Id_List){
    for(int i=0; i < personal_id_list.length(); i++){
        int digitNine = getDigitNumberNine(personal_Id_List.elementAt(i));
            if(digitNine % 2 == 0){
                print("The person is a female")
            }
           else if(digitNine %2 == 1){
                print(the person is a male)
           }
    }
}
```

6.2.2 Path testing

Path testing is a complementary method to both function- and structural testing, and verify how much of a program that is executed during testing. This measure is referred to as *coverage*. There are different types of path testing. Three of these are shown in the flowchart in *Figure 9*. Optimally all paths in a program should be executed. However this is not possible in some cases. For example a program with a for-loop, a program can then consist of an infinite number of paths. Still 100% coverage can be achieved when defined as: all independent paths in a program should be executed at least once.

The flowchart in *Figure 9* represents the code from Example 2 in the previous section, where if-statements are represented as nodes and branches show which paths that can be executed in the program. There are different strategies that can be used when analyzing which parts of a program that can be tested. Some of these strategies are repre-

sented in *Figure 9*. For further information about this type of testing we refer to [Fenton].



Branch (1): <1,2,3,6,4,5>
Statement (2): <1, 2, 3> <1,4,5>
All paths (n): <1,2,3(6, 4, 5)^n>

FIGURE 9. The figure shows different types of testing strategies that can be used during structural testing. The figures in the parentheses represent the minimal number of test cases necessary to fulfill strategy.

6.3 Static testing

Static testing differs from functional- and structural testing in that no execution of the program is needed, in order to locate defects. There are different types of static testing, but this report only covers *inspection techniques* (*Section 6.3.1*) and also describes a test method called *pair programming (Section 6.3.2)*, because a less formal version of inspection is integrated in the method and that U/PD is considering to introduce the method into their development process.

6.3.1 Inspection

Inspection is a way to detect defects early in a project. Used the right way it can result in shorter project time decreased costs and higher software quality [Gilb93]. The opinions about the advantages with inspections differ in research. Those in *favour* claim that inspection is the fastest and cheapest way to find defects, and those *opposing* claim that inspections cost too much, and that the process of reading documents is tedious and not motivating.

The most formal type of inspection consists of predefined phases. Documents are read by several developers, and found defects are categorized, logged and further discussed in a meeting. An example of this type of inspection is described below.

A review is a less formal inspection technique that does not necessarily consist of any phases or logging of found defects. An example of a review could be when a developer (A) hand over a code document to another developer (B), who reads through the code trying to find defects and return feedback to developer (A). The code has been reviewed but no logging of the found defects has been made.

Stepwise abstraction is another inspection technique developed by Linger [Linger79], and is also the technique used in the research experiments presented in *Chapter 7.0*. The concept of stepwise abstraction consists of that several programmers identify subprograms. They determine the functionality of the program and identify the subprogram's function in the complete program. Once the programmers have developed a complete picture of the whole program, it is compared with the original program specification to identify differences, which then are studied.

The following is an example of an inspection process [Gilb93]. The inspection process contains three phases:

- Initiation
- Checking
- Completion

Initiation:

The aim with this phase is to 1) identify documents that are suspected to contain defects or 2) where a defect would be devastating. A chosen inspection leader also referred to as the moderator performs this work. In order for the inspection to be successful, the inspection leader should have received proper training for the role.

The identified documents chosen for the inspection are complemented with other documents, identifying rules and standards. A checklist is created from these to help developers to follow them, aiming to achieve the purpose and correctness of the documents. These documents are then the issue for the inspection, referred to as *checking* performed by developers.

Checking:

The checking consists of two parts, the 1) *individual checking* and 2) *the logging meeting*. Each developer is assigned a role in order to have different focus when checking the document, hopefully finding different types of defects. It is emphasized that a checker never should try to inspect large documents at one time, this could increase the risk of missing a potential defect.

The *individual checking* is estimated to consume about 20-30% of the total inspection time and is performed by comparing the checklist towards the document that is inspected. Potential defects found are then graded in *minor* or *major* defects. The defects found during the checking are used as input to the *logging meeting*.

All checkers including the inspection leader participate at the logging meeting. The meeting is conducted in the form of a brainstorming, where the potential defects identified are noted. The logging meeting also aims to identify new potential defects. However it is emphasized that no long discussions take place during the meeting. These should be held afterwards, otherwise the meeting takes too long time.

The purpose with the meeting is strictly to identify and logg: 1) all potential defects found by each checker and during the logging meeting, 2) improvement suggestions and 3) raised questions.

Completion:

After the meeting the potential defects should be corrected and followed up, i.e. it should be assured that no new potential defects have been inserted during the correction. The corrections should be logged and then it is up to the moderator to decide if the

so-called *exit criteria* are met, i.e. the estimated remaining number of defects in the documents are acceptable.

6.3.2 Pair programming

Pair programming is a method used in Extreme Programming (XP) [Beck00]. A specific feature of pair programming is that most work is done in pairs, reviewing each other's code during the implementation. This method has proven to reduce the number of defects early in projects and increase software reliability [Williams00].

The thought behind XP is that two people stand a better chance to solve a problem than one. The theory is backed up with experiments, showing that developers using pair programming solve problems faster and with higher quality, i.e. develop software with less defects [Williams00].

There are not many rules for how pair programming should be performed. However there are some guidelines that should be followed.

1. Test cases shall be designed before the implementation.
2. All test cases shall be saved for later use.
3. There is a focus on simple solutions. If a simpler design solution is found during implementation it must be applied.
4. While one developer is programming, the other one is reviewing the code and at the same time critically trying to find flaws in the design.

Pair programming does not only help solving problems faster and increase software reliability, but has also been noted to increase developers confidence in their solutions and that they find it more "fun" to work [Williams00]. Further it is a great method for spreading knowledge between programmers.

Although the positive result of pair programming, there are some negative aspects with the method as well. For instance even though the development time is shorter, the total number of work hours is higher. This due to that two developers are working on the same task. However logically the cost in maintenance should be reduced due to fewer defects in the product. Unfortunately this relation has not been studied in any research that we are aware of.

Pair Programming at U/PD

The discussion about introducing pair programming at U/PD has risen before, but has not been applied as a standard in projects. This due to the lack of time for evaluating a new method and also that it seems difficult to argue for introducing pair programming in an organization. However assume that the advantages with, pair programming stated in previous section, are true. This would shorten the project time and improve the quality of the product, but to a price of increased total number of development hours excluding the hours spent on maintenance. This results in two questions:

1. How much is it worth to be able to deliver a product earlier?
2. What is the difference between the increased cost of pair programming before maintenance and the possible reduced cost of maintenance?

The relevance of the questions is high. However it will only lead to speculations unless the reduced development time and cost can be visualized. Assuming that the suggested test framework in *Chapter 10.0* was implemented at U/PD, it would provide a base for evaluation of the effectiveness of pair programming. Information that would be interesting to evaluate, is if pair programming reduces time and the number of defects compared to the previous process?

7.0 Test Technique Effectiveness (TTE)

This chapter relates to the fourth and fifth statements in the project hypotheses, i.e. there should be defined expectations on which test techniques that is best at detecting a certain defect and in which test phase. Most of the test techniques described in the previous chapter is used in research experiments in the TTE area. TTE is a measure of how well one or several test techniques are at detecting defects. TTE helps to provide better understanding of the test process in order to identify possible process improvements, and it is therefore of interest when developing a test model.

Several experiments have been made in the area and on parameters that can affect the TTE such as: program type and programmer experience. This is best visualized by describing the result of a few experiments, which is done in the following sections. However because of the circumstances of the experiments, it can be questioned if they can be compared and also if the conclusions are correct. These questions are raised based on three studies described in the following sections. For instance the studies evaluate slightly different techniques. Further all three studies used programmers with different programming experience. These are parameters that could affect the outcome of the study and also seem to be the case in the experiment performed by Basili and Selby [Basili87], where groups with students and professional programmers are compared. The following sections describe the three studies, highlighting the common conclusions and also the differences.

7.1 Combining software strategies: Selby

The first of the presented experiment was performed by Selby [Selby86]. 32 professional programmers participated in the experiment with the task to apply three test techniques on three different types of programs. The main focus of the study was to evaluate: 1) TTE by combining different techniques and 2) test technique efficiency.

The test techniques evaluated in the experiment are:

1. Code reading by stepwise abstraction.
2. Functional testing using equivalence partitioning and boundary value analysis.
3. Structural testing using 100% statement coverage.

The major result from the study was that 1) a combination of the three test techniques detected 17.7% more defects on the average than the three single techniques did. This was a 35.5% improvement in defect detection. 2) The most effective combinations for finding defects were when two programmers performed code reading or when combining a code reader and a functional tester. 3) When looking at how different experience levels affected the percentage of defect detection, the combination of two advanced programmers proved to be most effective and 4) the most efficient alternative was when code reading was performed by one experienced programmer.

Even though it is not considered one of the major results in the research, an interesting result is that the number of defects found was better when combining any two programmers, regardless of experience level, than only one advanced programmer, i.e. two junior programmers detected more defects than one advanced programmer did.

7.2 Comparing three test techniques: Basili and Selby

The following experiment was performed by Basili and Selby [Basili87]. They used 32 professional programmers and 42 advanced students in the experiment. The experiment consisted of testing four different programs, with the same test techniques as in the previous study, aiming to compare the aspects of: 1) defect detection effectiveness, 2) defect detection efficiency and 3) which type of defects that was found by a specific test technique. The experiment also aimed to evaluate how different parameters affected the result such as: 1) programmer experience and 2) program type.

The main conclusions drawn from the experiment were that 1) professional programmers using code reading, detected more defects and faster than with the other techniques. Further functional testing was better than structural testing, even though the defect detection rate did not differ. 2) In a comparison of the groups with advanced students there was no difference between the techniques except for one group where *function test* and *code reading* were better than *structural testing*. 3) It was also concluded that the number of defects found was dependant on the software. 4) Regarding the effectiveness of an individual test technique and certain defects, code reading detected more interface defects and 5) function test detected most control defects.

7.3 Comparing and combining software defect detecting techniques

The following study [Wood97] is a replication of a study that has been performed several times before, where the previous experiment being one of them. The testing was performed by 47 students. The task was to test three small C programs, within three hours. The study focused on four areas: 1) number of failures observed, 2) number of defects detected, 3) the effectiveness of each test technique in observing failures, and 4) how long it took to locate the defect.

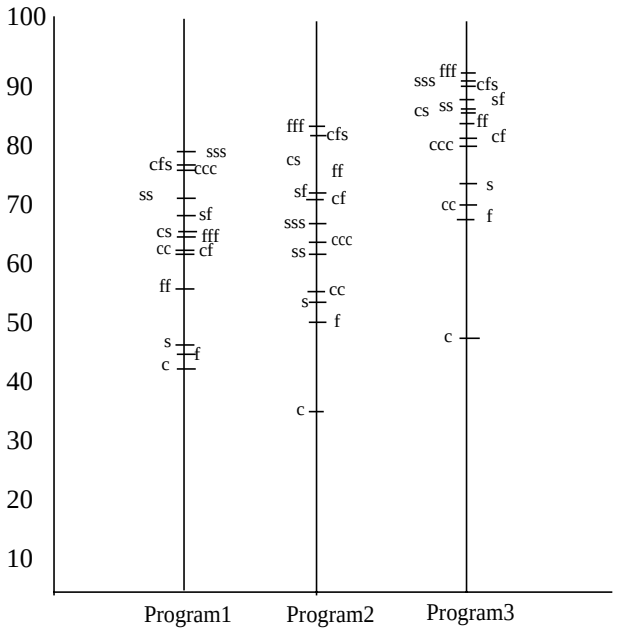
The test techniques evaluated are:

1. Code reading by stepwise abstraction.
2. Functional testing using equivalence partitioning and boundary value analysis.
3. Branch coverage.

The major conclusions drawn from the experiment were 1) no single test technique was found to be better than any other in the aspect of TTE. 2) Further no technique was best at detecting a certain type of defect. Wood concludes that these parameters are dependant on the program type. 3) It is further stated that the best way of testing is a combination of the techniques. The experiments showed that no single technique detected

more defects than a mixed pair, and no mixed pair detected more defects than a combination of three techniques.

The TTE when combining several test techniques and how these depend on program type, is showed in *Figure 10*. The combinations of test techniques consist of one to three techniques. Each technique is represented with a character s (structural tester), c (code reader), and f (function tester). For example the combination “cfs”, means that three programmers performing different tests, i.e. code reading, function-, and structural testing. Another example “fff“ means that all three programmers perform function test.



c = one code reader
s = one structural tester
f = one functional tester

FIGURE 10. The percentage of defects detected with different combinations of test techniques.

We would like to further emphasize that the experiment results could be an affect of circumstances during which they where performed. Wood commented that the students expressed disappointment with the number of defects found and states *“one picture which does build up quite clearly in all the experiments is that all the techniques (or all the testers!) are quite poor”*. This statement indicates that the result could have been greatly affected by the lack of the students’ experience level. It would be of great interest to evaluate how well the testing was performed, i.e. where all the domain- and boundary values identified? Otherwise the conclusion is based on the skills of the programmers, rather than the effectiveness of the test techniques.

It should also be noted that this study shows that there is no test technique that is better at detecting certain types of defects, which also was concluded by Basili and Selby in the student experiment, but not for the experiment with professional programmers. Further Wood evaluates structural branch coverage where as the other studies evaluate structural testing using 100% statement coverage, which also could have affected the result of the experiment.

8.0 Orthogonal Defect Classification (ODC)

Even though really good conclusions could not be established between test techniques and defect types in the previous chapter, this is done in the test framework ODC, which also has proven to work in practise. Defect classification is the corner stone in ODC in order to evaluate the test process, which is also is supported by other researchers. Defect classification can be used for improved understanding of the development process and to evaluate the effectiveness of test techniques. *Counts of defects found during the various defect detection activities in software projects and their classification provide a basis for product quality evaluation and process improvement* [EmWi98]. Therefore when planning to make changes in the development process it is essential that the current development process is understood, i.e. can be visualized by measurements based on defects.

Unfortunately there are some problems related to defect classification. The main problem of focus in this report is to minimize the risk for misunderstanding of different defect types. The area is referred to as *repeatability* and means that the risk for two developers to categorize a defect under different defect types is minimized. If repeatability is not achieved, there are reasons for justifiable doubts in decisions based on statistics drawn from these [EmWi98].

The problems stated are addressed in the ODC framework, which is a defect classification framework developed by IBM [Chillarege92]. The overall goal of ODC is to make the development process manageable and controllable by: 1) identify problems in both the development process (*defect types Section 8.2*) and verification process (***defect triggers*** *Section 8.3*). Further ODC mathematically visualizes the relation between defects and the development process (*Section 8.4*). These insights are naturally desirable for any company that wants to improve there development process, but has according to Chillarege not been possible to achieve until the ODC was developed.

8.1 The basic thought behind ODC

The ODC aims to achieve these goals by minimizing the gap between 1) *statical defect models* and 2) *casual analysis*. Chillarege describes the gap between the two areas as: "*in a large development effort it is akin to studying the ocean floor with a microscope*". By this he means that *statical defect models* step back from the development process trying to mathematically predict the product reliability independent of defect types. The negative aspect of this type of models is that the knowledge is provided late in the project cycle and that the information cannot be used to identify process problems in greater detail, but serves more as a confirmation of the software reliability. *Casual analysis* on the other hand is much more process related, aiming to analyze and provide developer feedback on each occurred defect. However there cannot be any evaluation on which effect changes based on such analysis has on the development process as a whole. The reason is that there is no mapping between the development process and the product reliability.

The mechanism used in ODC, referred to as the cause-effect relationship, narrows the gap between the two areas. This practically means that defect types are mapped towards a specific point in the development process, which makes it possible to identify when and where a defect was inserted. This makes it possible to predict the product reliability as well as provide developer feedback, even if the information is not as detailed as with causal analysis.

8.2 Defect type

The defect types in ODC provides a basis for identification of problems in the development process. In order to achieve a repeatable defect classification that is mapped to the development process, the defect types should not only be easy to understand, but also cover all possible defects types that can occur in a product. Further each defect type should be mapped to a particular phase in the development process. Because of these requirements, the design of defect types in ODC must fulfill: 1) simplicity (*Section 8.2.1*), 2) the sufficient condition (*Section 8.2.2*) and 3) the necessary conditions (*Section 8.2.3*). Also for each detected defect it should be decided if the defect was a result of missing or incorrect code. For example in case of an interface defect, are parameters missing or are they incorrect?

8.2.1 Simplicity

Simplicity relates to the previously described defect classification problem, i.e. repeatability. The ODC strongly emphasizes that the defect classification is simple. Chillarege uses the description "*the defect types should be obvious to the programmer, not leaving room for much confusion*". If this is obtained it fulfills the requirement of a repeatable defect classification. One of the actions taken in ODC to fulfill this requirement is the decision of only using eight defect types, listed in *Table 1*. These are selected to be general enough to apply for any software product. Further the granularity of the defect types should be so that it does not matter in which phase the defect is detected, it should still map to the same point in the development process.

Defect Type	Description
Function	Defect that affects the capability of a program such as end-user- and product interface.
Interface	Defect between interacting components or modules.
Checking	Defect related to program logic that has failed to validate data before usage.
Assignment	Defect related to a few lines in the code, such as initialization or data structure.
Timing/serialization	A defect related to shared and real-time resources.
Build/package/merge	A defect due to a library systems, configuration management or version control.
Documentation	An error that affects publication or maintenance notes
Algorithm	Defect that affect efficiency or correctness, that affect the task, although the design is correct.

TABLE 1. A description of the ODC defect categories

The defect classification is also required to fulfill the *sufficient*- and *necessary condition*s, described in the next two sections. Fulfilled conditions assure that there are enough definitions of defect types to cover all occurrences of possible defects (*sufficient condition*) and that the identified relations are uniquely mapped to a project phase (*necessary condition*). Although the sufficient condition means that the identified defect types must have subtypes, the focus is on the eight predefined defect types (*Table 1*), which is the level where the mapping is made towards project phases. These mappings are described in *Section 8.2.3*.

8.2.2 The sufficient condition

The sufficient condition relates to the fact that it is not enough with eight defect types to describe all possible defects that can occur. Conclusively each defect type must contain sub defect types, referred to as elements in ODC. The aim is to identify a set of elements that spans over the whole test process. However to develop a defect classification that meet the sufficient condition is difficult, and should be viewed as an evolving process. Chillarege means that the validation of the correctness of the classification and performed measurements improves with increased experience.

8.2.3 The necessary condition

Provided that the defect types are repeatable and that the sufficient condition is considered, the next area to analyze is the necessary condition. The aim of the *necessary condition* is twofold, by introducing a relation between defect types and the project phases it is possible to:

1. limit the negative effects of opinion-based classification, and
2. pinpoint process related problems.

The problem with defect classification does not necessarily have to be the classification itself. Many problems are due to the human factor. Chillarege refers to two types of classifications: *1) opinion-based*- and 2) *semantic classification*. Opinion-based defect classification is error-prone, making decisions based on this material unreliable [Chillarege92]. The opinion-based classification is typically characterized by letting the developer answer the question, *where was the defect injected?* Semantic classification on the other hand is defined by the semantics of the fix. Research has proven that by uniquely relating defects to a semantic defect type, it is possible to identify changes in the product reliability to a specific type.

The necessary condition tries to reduce the negative influence of the human factor, and introduce predefined relations between defect types and process phases. The goal of this requirement is to identify process related problems based on occurred defects. The relation between defects and process phases makes the identification of the defect type sufficient in order to identify in which phase the defect was injected. *Table 2* shows the

predefined orthogonal relations in ODC, where the developer is restricted to only identify the defect type.

Defect Type	Process Associations
Function	Design
Interface	Low Level Design (LLD)
Checking	LLD or Code
Assignment	Code
Timing/Serialization	LLD
Build/Package/Merge	Library Tools
Documentation	Publications
Algorithm	LLD

TABLE 2. The mapping between defect types and associated process activities

8.3 Defect triggers

The previous discussions about ODC has been concerning identification of problem areas in the development process, defect triggers evaluate how well the verification process is working. A *defect trigger* can be defined as the event that reveals the existence of a defect due to the current circumstances such as: product- and development environment.

Although increased understanding of the product environment can help to improve the test process, this area is not discussed in greater detail in this report, much due to the complexity of the area. Further defect triggers are not described in detail in the ODC framework. Instead defect triggers are discussed in general terms, referring to project examples. One of these examples is described below.

Example 3: A defect trigger analysis was made on the MVS operating system at IBM, where storage corruption was the defect in focus. At first *timing* was believed to be the primary trigger for these defects. After an analysis it showed that timing only triggered 12.4% of the storage corruption defects and that the main cause of the problem was boundary conditions. Before the defect analysis was performed, the variation of hardware during test was the most logical choice in order to detect these types of defects earlier. However the analysis showed that inspections would uncover more defects to the same or less cost.

8.4 Identifying process problems with ODC

We have now reached the point where it is possible to describe how process problems can be identified based on ODC. Because of that it is not always easy to make an analysis on just defect data, ODC applies two ways of visualizing process problems. One is drawn directly from the ODC framework (*Section 8.4.1*) and compares how effective the different test phases and test techniques are at detecting defects compared to what is

expected. The other way is a result by combining ODC and SRGM (*Section 8.4.2*) and is able to pinpoint process problems related to a specific defect type.

8.4.1 Comparing expected behavior with actual result

The cause-effect relationship between defects and project phases can be used to identify process problems, but so far it has not been described how. It is not enough to just count the number of occurring defects in order to identify the source of a process problem, one of the reasons is that a defect can potentially be caught by different test techniques. Instead ODC aims to compare how well test techniques live up to their expectations. The evaluation process is supported by 1) a principal association table (PAT) (*Table 3*), describing predefined relations between defects and test techniques, and 2) a binary tree, visualizing how well test techniques fulfill expected test result.

The use of these methods demands that there is an understanding, or belief, for what is expected from the different test stages and test techniques. A PAT is built by setting the different test stages and test techniques on the vertical line in *Table 3*, and the different defect types on the horizontal line. The next task is to identify the relations between the defect types and test techniques, showing which defect types that are believed to be detected in a specific test stage during testing.

The principal association table:

Stage/Defect Type	Function	Checking	Timing	Algorithm
Design	X			
LLD			X	
Code		X		X
HLD Insp.	X			
LLD Insp.			X	
Code Insp.		X		X
Unit Test		X		X
Function Test	X			X
System Test			X	

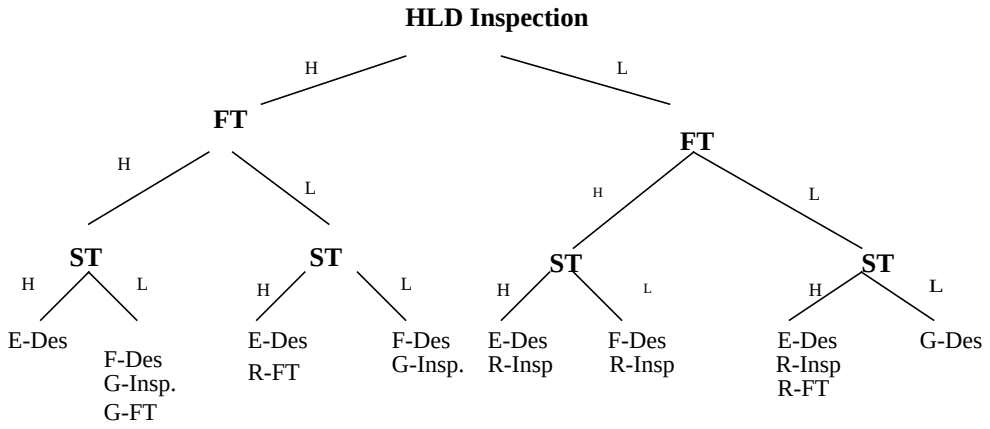
TABLE 3. The principal association table shows where and how in the development process different defect types are expected to be detected.

The next part of the evaluation process consists of building a binary tree, which contains the different test stages in the vertical line of the PAT. This tree is just a simple way to confirm if the test techniques detect defects as expected. For instance the binary tree in *Figure 11* is built by extracting the test stages expected to detect defects of type function, from *Table 3*.

The root of the tree represents the first verification stage and the following stages are represented in the following levels in the tree, i.e. Function Test and System Test. Even though System Test is not the primary test technique to detect defects of type function,

since these should have been detected earlier in the test process, it has the characteristics for doing so. However if too many defects of type function is detected in System Test it would mean that the previous test techniques, expected to detect them, has failed and that a process problem is identified.

After each level of verification it should be decided if the number of defects is higher (H) or lower (L) than expected. For example the sequence H-L-H in the *Figure 11*, indicates that there is a gap between Function Test and System Test. It seems like System Test has to detect the defects that should have been caught earlier in the test process. An analysis should be made on how improvements could be made in the future in order to prevent the scenario from repeating itself.



Design status: E: Exposed, F: Fixed, G: Good, R: Revamp

Detected defects: H: High, L: Low

Verification stages: HLD: High Level Design, FT: Function Test: ST: System Test

FIGURE 11. The binary tree representing different test stages, where HLD Inspections and Function Test are expected to detect most defects of the type function.

8.4.2 Identifying risks combining ODC with SRGM

Unlike the previous section, evaluating performed testing, this section predicts future test result by combining ODC with SRGM. SRGM is used in Software Engineering to assess and predict the reliability of a product and is therefore an excellent management tool. Unfortunately the drawback with this technique is that valuable information is not provided until a later stage of a project cycle. Further it is hard to pinpoint a process problem with this technique.

By using the characteristics of ODC, dividing occurred defects into groups, the overall growth curve could be divided into several corresponding curves. Each curve would then represent a defect group, which provides better process insight. This insight is a result of the cause-effect relationship defined in ODC, where each defect type is mapped to a process phase. By analyzing and comparing the different curves, it is pos-

sible to see which defect types that indicates a process problem, and conclusively identify which part of the development process that should be improved.

How this is done is best showed by an example the report [Chillarege92]. The example is taken from a large project that used ODC. The data is retrieved during System test, and the curve in *Figure 12* shows the cumulative number of all defects. Judging by the curve, the project has been going on for 100 days and the number of defects is still increasing. The prediction indicates that the number of defects will not start to recede until the project has reached about 170 days, marked (π). However the curve provides no insight of which defect types that are most problems for the product and therefore no decisions can be made on where most effort is needed.

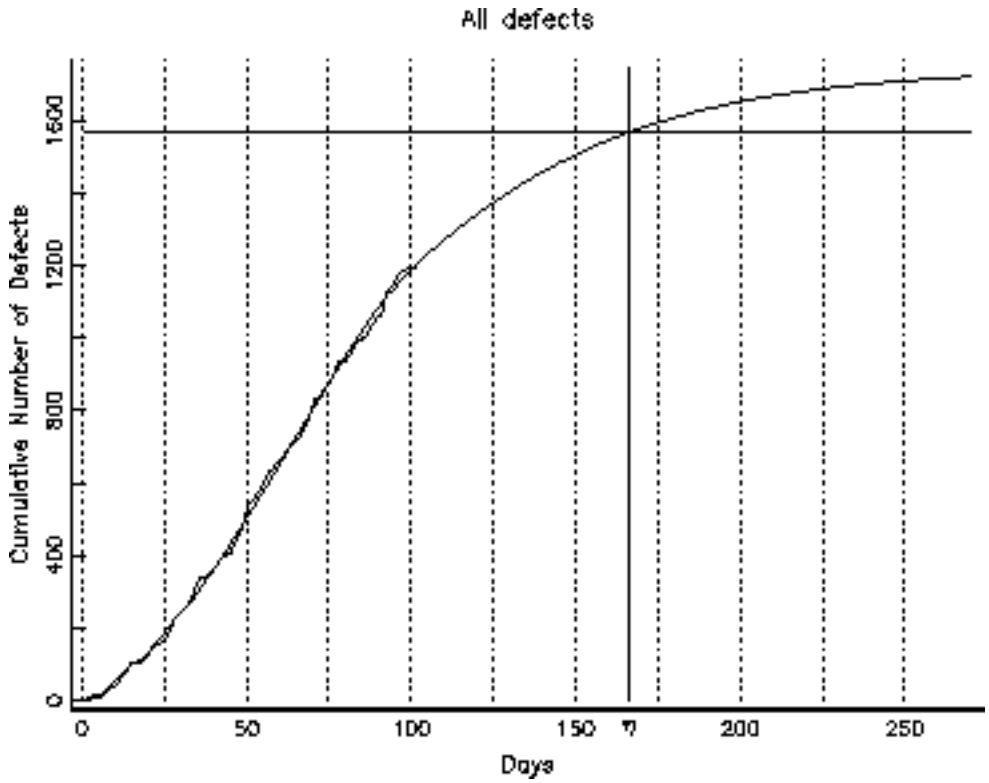


FIGURE 12. A growth curve of the cumulative number of all defect types. The figure is copied from the report [Chillarege92]

In *Figure 13*, the overall growth curve has been divided into tree curves, one for each defect group. According to Chillarege, it is sometimes better to collapse some categories, to reflect the development process better. This is done in this example where the three curves in the figure represents the following defect groups:

1. Function
2. Assignment/Checking
3. Miscellaneous

The figure shows that both function and assignment/checking defects are stabilizing, where as the other defects still are increasing. Since defects of type function and assignment/checking relate to the design phase, this indicates that the product designers can be reallocated to where their skills serve a better purpose.

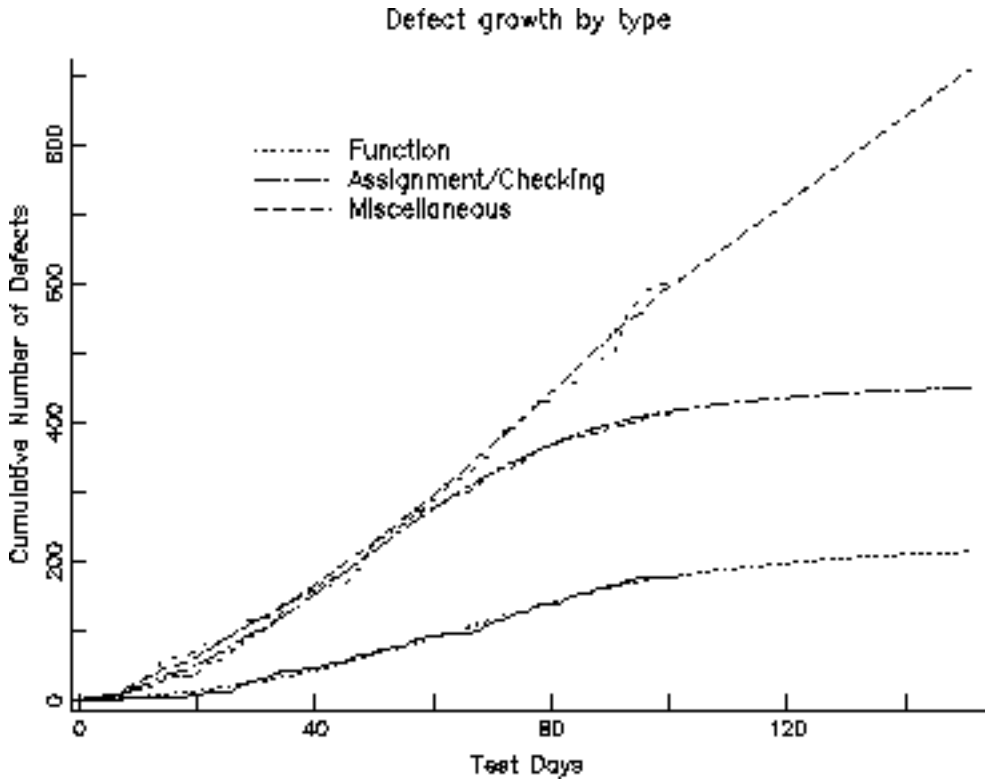


FIGURE 13. The growth curve in *Figure 12* has been divided in several curves where each curve represents one or several defect types. The figure is copied from the report [Chillarege92].

9.0 U/PD's Test Environment

The test framework developed in the report is based on ODC and TTE. However in order to adapt the framework to the U/PD environment, knowledge is required about the specific product- and development environment. This chapter describes the specific test techniques currently used at U/PD, and also describes the current defect categorization in the light of possible usage for test process evaluation.

9.1 Defect classification at U/PD

The defect types used at U/PD are listed in the so called Trouble Report Tool (TR-Tool). When a failure occurs in Function Test, a description of the failure is logged and sent to the developer who is most likely to identify the defect. The developer is responsible for the identification of the defect and also for describing how the defect can be corrected. The aim of this procedure is to assure that the defect finally is corrected and also that the total number of defects is logged.

It should be noted that the identification and logging of a defect type is not used for any process evaluation, but only serves as a reference to a defect. As a matter of a fact the current defect classification cannot be used for a process evaluation. The reason is that it suffers from two problems, which would result in unreliable statistics if such were extracted from the occurred defects:

1. The defect classification is designed for another product. Which results in that all types of defects that can occur in the SDP is not represented, whereas some defects that cannot occur in SDP are.
2. The classification is ambiguous and overlapping, which increases the risk of two developers categorizing a defect under different categories.

An example of this problem is showed by comparing two of the defect types defined in the TR-Tool: 1) Coding fault and 2) Incorrect handling of variable or symbol. We interpret coding fault to be a defect in the code. This is a rather wide concept and covers many of the defect types that can occur in a program, including the defect type "Incorrect handling of a variable or symbol".

The example visualizes the problems listed above and indicates that no information drawn from the occurred defects can be used to evaluate the development- or test process. The highlighted problems show that a new defect classification scheme is needed in order to evaluate the current test process at U/PD and is taken into account in the suggested test framework in *Chapter 10.0*. In the framework, each defect type is mapped to one or several test techniques, which are not all described in ODC. Therefore the next section describes which test techniques that are used at U/PD.

9.2 Test techniques used at U/PD

The following sections provide a brief overview of the different test techniques used at U/PD, which also are referred to in the developed framework. Comments regarding how the tests are performed is based on previous chapters and the survey performed at U/PD (*Chapter 14.1*).

9.2.1 Function- and structural test

The developers design test cases for both function- and structural test as described in *Chapter 6.0*. Although the test cases are designed, they are not designed during optimal circumstances. *Section 7.1* describes the importance of designing test cases in several stages and to consider both domain- and boundary values. According to the survey most of the developers designs the test cases during the implementation phase. However test cases are not designed based on the requirements specification or the program design, which increases the risk of missing test cases. Further the developers are not being consequent when analyzing domain- and boundary values, which also might result in missed test cases.

9.2.2 Reviews

Formal inspections are not performed at U/PD, instead code reviews are used aiming to optimize algorithms, improve code structure and program design. Further there is a specific focus on memory management. There is no description of when to perform the reviews, instead this is decided by each developer. The choice of reviewers is decided by the author of the code. The aim is to have two or three persons to review the code, where at least one of the developers is an experienced developer.

9.2.3 Tools

During the execution of test cases there are several tools used, for instance tools for: memory leaks, coverage and performance. Although it is seen by the management, that these tools are used frequently, this is not the case. According to the survey and from discussions with developers, the tool used for checking memory leaks is the only tool used frequently. The reason for this seems to be the lack of education on the tools.

9.2.4 Autotest

U/PD uses a form of testing referred to as autotest. There are two types of autotests used, both during unit test. One of the tests is used for testing a specific class, where as the other is used for testing larger parts of the system. The concept autotest makes it possible to test specific parts of the system, independent of the rest of the system. When the environment for the autotest is implemented it remains to identify test cases, which is executed each time that specific part of the system is tested. Test cases are analyzed each time any changes are made in the program. Test cases are then added, changed or removed.

10.0 A Test Framework Combining ODC and TTE

Up till now the report has covered the base for the framework described in this chapter, i.e. ODC, TTE and the current test environment at U/PD. This chapter puts all the pieces together and describes the resulting framework. A short description of a test model is also included in the solution, but does not cover the complete test process at U/PD (*Section 10.1*). Instead the focus is on the framework that can evaluate the test process and provide information for future test process improvements.

The framework provides value in three areas, firstly a base for evaluating the current test process compared to expected test result. Secondly the framework has the advantage of providing developer- and management feedback, describing if enough testing has been performed. Finally the last part of the framework provide information by combining ODC and SRGM, which can be used to identify defects that are particularly problematic.

The development of the test evaluation framework consists of several stages that should be implemented in certain order, since each stage depend on the previous one. The first phase is the design of a defect categorization scheme (*Section 10.2*), since this is considered to be the base for process- and quality improvement [EmWi98]. The second stage of the implementation consists of developing a PAT containing the test techniques used in testing (*Section 10.3*), which are related to the identified defect categories. The next stage consists of developing checklists in order to logg performed testing and occurred defects (*Section 10.4*), and the final stage of the test framework implementation consists of developing a tool that can provide developer- and manager feedback, also based on the performed testing and occurred defects (*Section 10.4*).

10.1 The test model

This project has not resulted in a complete test model, since it was concluded that this is a difficult task, due to that research result within TTE area is contradicting. However a test model can be developed with the help of this report and then evaluated by the developed test framework.

A test model document should optimally be developed by U/PD based on the test techniques listed in the PAT, described in *Section 10.4.1*. Further a description of all the test techniques should be given and also when they are expected to be used, i.e. in which test phase. Several parts of this report can be extracted for such a document such as: Which test techniques should be used? How and when test cases should be designed? How pair programming should be performed?

Figure 14 gives an overview of when different test techniques are expected to be used in a specific test phase. For instance the design of test cases for function test (TCD FT), should be extracted from the requirement specification and design document, where as the test case design for structural test (TCD ST), should be performed during and after implementation. This type of analysis relates to the project hypotheses 5, i.e expectations of when certain defects are expected to be detected and with which test technique.

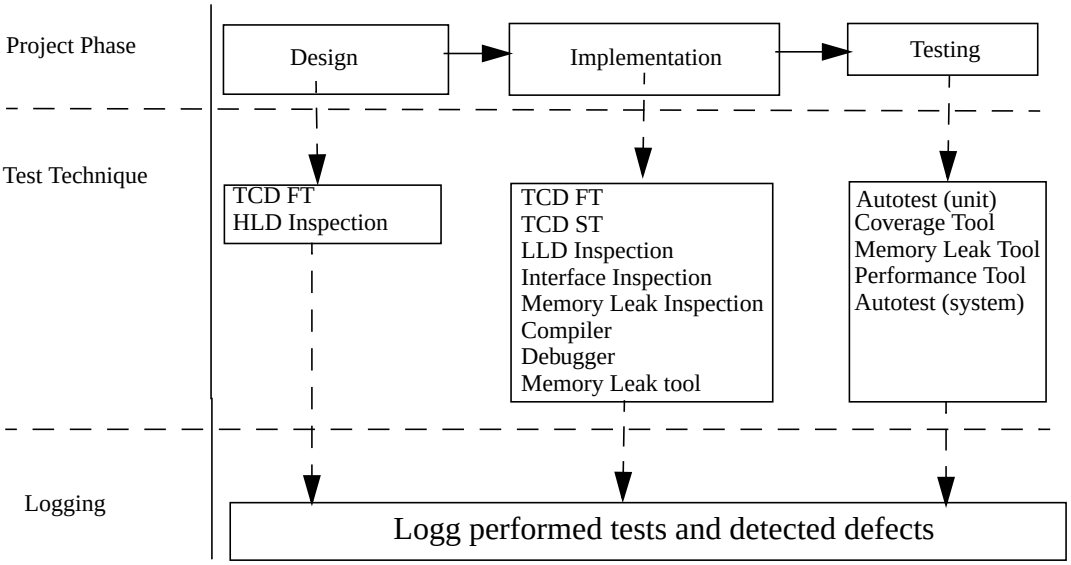


FIGURE 14. An overview of when different test techniques should be used.

10.2 Defect classification

The first step when implementing the developed framework is to design a defect classification, which relates to the two first statements in the project hypotheses. As described previously in the report a defect classification is the base for process evaluation and -improvement. Since the defect classification currently used at U/PD is not repeatable, a new defect classification should be developed that fulfills the requirement of repeatability in order to evaluate the current test model.

A suggestion of a new defect classification scheme is given in *Figure 15*, containing the ODC categories in the left column and the SDP specific defects in the column to the right, referred to as elements in ODC. Besides the ODC defect categories and the SDP specific elements, a new column is added containing sub-categories of the ODC defect categories. The purpose of adding sub-categories is to make the categorization of a defect easier for the developers, which also is one of the aim with the ODC categories. For instance, the elements in the defect category *Assignment* in *Figure 15* can be divided into three sub-categories: 1) Memory handling, 2) Calculation, and 3) Variable initialization. It should be noted that it is equally important that the sub-categories and identified elements are repeatable, otherwise the defect categorization scheme suffers from the problems that should be avoided, as described in *Chapter 8.0*.

It should also be noted that the elements is not general enough to be used for any other product, but has to be identified for each product environment. This is a problem for Ericsson Software Technology, since most departments use the TR-Tool. If the framework is to be used by all the departments, a change in the defect classification will affect other departments as well. Conclusively an analysis of a defect classification has to be made for each development- and product environment, in order to achieve a repeatable defect classification that fulfills the ODC conditions: simplicity, sufficient and necessary conditions. This also requires an administration tool that can add, remove and change the defect elements.

Note that the defect classification in *Figure 15* consists of the term *Other*, which means that 1) there has not been more identified sub-category or 2) more classes or elements can be identified depending on the development- and product environment. When performing an analysis on which defect classes and elements that can occur in a product, it is equally important to identify classes and elements that should be removed, in order to fulfill the conditions described in ODC (*Chapter 8.0)*.

Defect Type	Defect Sub-categories	Defect (Elements)
Function	• Specification • Other	• Wrongly specified function • Incorrect interpretation of requirement • Other
Assignment	• Memory Handling • Calculation • Variable handling • Other	• Memory Leak • Pointer Management • Copying overflow • Arithmetic miscalculation • Uninitialized variable • Other
Interface	• Low Level Interface • External Product Interface • Other	• Wrong number of parameters • Wrong type of parameters • Other
Checking	• Other	• Statement Logic
Timing/Serialization	• Other	• Deadlock • Livelock • Other
Build/Package/merge	• Other	• Incorrect revision state • Incorrect article number • Component Installation • Other
Documentation	• Other	• Other
Algorithm	• Other	• Performance • Other

FIGURE 15. The suggested defect classification and elements for SDP.

10.3 A PAT adapted to the product- and development environment

The second step of the implementation of the new framework, is to construct a PAT that consists of all the test techniques used in the U/PD's test environment, further each test technique should be mapped to one or several of the ODC defect categories in *Figure 15*. This step related to the project hypotheses 4, making it possible to provide test process feedback on which defects that is the result of poorly performed testing.

Test techniques / Defect types	Function	Checking	Interface	Timing	Assignment	Algorithm
Pair Programming	X	X	X	X	X	X
Function Test	X	X				X
Structural Test		X				X
HLD Inspection	X					
LLD Inspection			*X*	X		
Interface Inspection			*X*			
Memory Leak Inspection					*X*	
Performance Inspection						*X*
Compiler		*X*	*X*		*X*	
Memory Leak Tool					*X*	
Performance Tool						*X*
Autotest (Unit)		*X*				*X*
Autotest (System)	*X*			*X*		*X*

FIGURE 16. A PAT combining the PAT from ODC with test techniques used at U/PD.

The suggested PAT is adapted for the U/PD's test environment, where the test techniques is listed in the left column and the defect categories is listed in the top-row, see *Figure 16*. The major difference between the suggested PAT in *Figure 16* and the one described in *Section 8.4.1*, is the level of detail. Most test techniques used at U/PD is included in *Figure 16*, whereas in ODC the test stages is on a much higher level, containing test phases and test technique areas such as code inspection. An example of the increased level of detail in *Figure 16* is that code inspection has been divided into three other inspection techniques, i.e. memory leak inspection, interface inspection and performance inspection.

The reason for breaking down the PAT on a more detailed level is to provide more detailed process feedback. For example, consider a developer that is testing his/her code, using *interface inspection* and *performance inspection*. If the PAT described in ODC where to be used for logging performed testing, the performed inspection should be logged as code inspection. Since the developer did not perform any memory leak inspection, the risk of missing memory leaks has increased. However it is difficult to evaluate where test process improvements can be made, since it is possible to assume that memory leak inspection has been performed, because it is a type of code inspection.

Further the PAT consists of test techniques such as pair programming and test tools. Test tools are not included in ODC, but since they are used at U/PD these should also be included in the evaluation. Some of the tools currently listed in PAT are: Compiler, Performance Tool, Memory Leak Tool. Further pair programming is included in the PAT. This is not a test technique, but rather a method. Therefore it can be questioned if it should be included in the PAT. However there are several parameters that can affect the test result and should be logged in some way, in order to understand the test result, where pair programming is one of these parameters. Similar parameters that could be logged is mentioned in *Section 12.3*.

The final issue related to the content of the PAT regards the fact that research results from TTE was supposed to validate the relations between the test techniques and the defect categories. However because of the contradictions between the research results in the area, this has not been done. Still based on the conclusions from Basili (1987), that was performed on professional programmers, interface inspections was concluded to be the best at detecting interface defects, which has been considered when designing the PAT. Further it should be noted that no matter which test technique that is best at detecting a certain defect type, several test techniques should be used in order to detect most defects, which also is concluded in research [Selby86, Wood97]

10.4 Test process evaluation based on logged testing

Once a defect classification scheme has been developed and expectations on test techniques has been defined in the PAT, it is possible to logg information of occurred defects and performed testing and draw conclusions from these. The third step of the test evaluation framework consist of developing two checklists, for the logging of which test techniques that is used during test, listed in the PAT (*Figure 16)*, and for categorizing occurred defects according to the defect classification in *Figure 15*. This step is a result of the project hypotheses 3 and makes it possible to perform the test process evaluation described in the next sections.

The logging of defects and performed testing makes it possible to: 1) provide developer- and manager feedback on how well testing has been performed (*Section 10.4.1*), 2) evaluate how well performed testing fulfills predefined expectations (*Section 10.4.2*), and 3) identify the most problematic defects during a project (*Section 10.4.3*).

When the logging has been made it is possible to perform the test process evaluation. The fourth step of the implementation of the framework consists of developing a tool that can provide test process feedback, based on performed logging, on both developer- and manager level. *Section 10.4.1* describes which type of feedback that could be provided on the different levels. The feedback differs from the one provided by ODC in that it is on a more detailed level and that it is not only given on manager level, but also directly to the developers. The last two sections (*Section 10.4.2* and *Section 10.4.3*) are practically the same solution as described in the ODC framework, the extra value provided by this report regards possible automation of the two areas: binary tree and SRGM.

10.4.1 Feedback on performed testing

Feedback based on extracted project data is an excellent way to evaluate if testing has been performed thoroughly enough. The feedback can provide information both to developers and managers. The developer feedback is based on how the developer has performed testing, i.e. has all the test techniques been used according to the test model? Since the occurred defects are logged, it can easily be evaluated how defects can be prevented depending on which test techniques that have not been performed.

Feedback on performed testing is simply done by comparing logged information of performed testing and occurred defects, with the relations between test technique and defect types in the PAT. For example lets assume that the PAT in *Figure 17* is the result from the logging of performed testing. The highlighted test techniques *Function Test* and *Interface Inspection* have not been performed by the developer. It can then easily be understood that the risk for defects of the type Checking, Algorithm and Interface is higher than if the tests had been performed. If this is the case, can be concluded after performed Function- and System Test. If defects occur, of the type previously mentioned, the feedback to the developer will be that Function Test and Interface Inspection could have prevented these defects. By providing statistics to the manager of how testing is performed in general, related to most occurring defect types, the information can be used to improve the test process.

Class level	**Function**	**Checking**	**Interface**	**Algorithm**	**Timing**	**Assignment**
Function Test	X	X		X		
Structural Test		X		X		
Interface Inspection			X			
Memory Leak Inspection						X
Autotest (Unit)		X		X		

FIGURE 17. An extracted part of the PAT from *Figure 15* showing how the result performed testing can be analyzed.

The feedback should be personal aiming to remind the developer of how testing should be performed. Still it could be of interest for managers to receive some information from this feedback and compare with occurred defects. For instance: 1) How well is testing performed in general? 2) The difference between developers on junior- and senior level in detecting defects? How long time does an inspection take in relation to occurred defects? etc. This type of information is not provided by the developed test framework, however it is recommended that these type of parameters are identified by U/PD in order to improve the process evaluation. The next two sections describes how more information can be provided on manager level by using the binary tree and SRGM, described in *Chapter 8.0*.

10.4.2 Expectations on test result

The next step in implementing the test framework is to develop binary trees that can provide manager feedback of how well the current test model fulfills the management

expectations. The construction of binary trees for different defect types results in several binary trees and can be quite time consuming to develop, which is why we suggest that this activity should be automated. This task can simply be done by automatically build the trees from the PAT, where one tree represent one defect category in the PAT and the test stages are extracted from the defect category's column. Further, instead of grading the number of detected defects with high or low, as done in *Figure 11*, an interval should be defined representing the percentage of defects which a test technique is expected to detect.

For instance, lets assume that the Memory Leak Tool is expected to detect between 30 - 50 percent of defects of the type *memory leak*. A percentage of defect detection higher or lower that this interval should automatically lead to an evaluation of the test process, the principle is visualized in *Figure 18*. If in practice only 20% of the function defects are detected with the Memory Leak Tool, it can be questioned if the tool has been used thoroughly enough. Similarly if 80% of memory leaks are found with the Memory Leak Tool, it could indicate that something is wrong in the development process. For instance the increased number of detected memory leaks might be a result of a developer that has limited experienced of C++. An example of what could be done to prevent this, is that developers with limited experience of C++ should have their code inspected by a senior developer, with the focus on memory leaks. This type of feedback is difficult to obtain with the binary tree described in ODC, since as long as the defect detection is high, no process analysis is made.

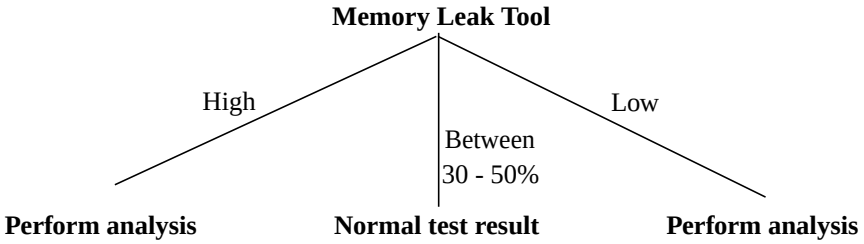


FIGURE 18. The figure shows that an analysis should be made if the number of detected defects is higher or lower that expected.

10.4.3 Identification of problematic defects

The final part of the test framework consists of SRGM that provides management feedback of process problems related to one or several defect types. The SRGM can be used as described in ODC, see *Section 8.4*. If some defects prove to be more problematic than others, i.e. is more occurring, it indicates that there is a problem in the development process.

ODC combined with SRGM can provide valuable information to project managers as decision material for reallocating resources. For instance, as described in *Section 8.4.2*, where it is concluded that designers can be reallocated due to that defects related to the project design is stabilizing. However we would like to extend the value of SRGM further to fit the idea of our framework better, which is focusing on process feedback on a more detailed level than ODC. We believe that the SRGM can be used to visualize the

software reliability on a defect element level. So, just like the overall growth curve is divided into different defect category curves in *Section 8.4.2*, each defect category curve could be divided into several defect element curves.This means that even though a defect category is stabilizing, there can be a specific type of element that should receive extra attention. The idea is exactly the same as described in *Section 8.4.2*, but can provide more detailed information on each defect element.

11.0 Conclusions

The goal of this report was to develop a test model and to improve the test awareness at the department U/PD, at Ericsson Software Technology in Karlskrona. Early in the project it was understood that it is difficult to evaluate the current test process at U/PD. The major reason for this was that 1) there is no repeatable defect classification, and 2) there is no information of which tests that is performed. This means that it is very difficult to evaluate which type of defects that occurs in the SDP and why they are not caught as expected.

Further it was found that it is hard to develop a test model based on research results, since the results heavily depend on the development- and product environment and has therefore resulted in different conclusions between research experiments. This insight resulted in another project goal, i.e. to develop a framework that could evaluate the current test process, providing decision material for possible process improvement possibilities. Although the base of a test model for U/PD is described in the report, the focus is on the test evaluation framework, since there is no assurance that the developed test model is optimal for the SDP environment. The two areas that were believed to provide most value to such a framework are:

- **ODC**: that consist of 1) a repeatable defect classification, 2) a mapping between defect types and test techniques, and 3) a method for evaluating how well the current test process fulfills the expectations of detecting defects.
- **TTE**: is a research area that evaluates which test techniques that is best at detecting defects. In relation to this area, research has been made on which test technique that is best at detecting a certain type of defects.

11.1 Hypotheses validation

The project hypotheses (*Chapter 3.0*) was stated in order to achieve the project aim of: 1) Evaluate if the current test model is followed, for example are all test techniques used according to the test model? 2) Evaluate how well testing fulfill predefined expectations, i.e. is a defect detected with the expected test technique and in the expected test phase? 3) Identify if there are any problematic defects that should receive extra attention during a project such as, if one or several defect types occurs more frequently than others?

The hypotheses is supported in research. The *two first statements* are validated by both Chillarege (1992) and Emam (1998), that emphasize the importance of a repeatable defect category in order to perform any process evaluation. The third statement, i.e. logging is necessary in order to evaluate the test process, is indirectly validated by the two first statements, since no evaluation can be made unless project data is extracted and logged in some way. The next two statements (4-5), i.e. predefined relations between project phases and defect types, and also predefined expectations on test phases, are related to the first phase of the project where we tried to identify a relation between occurred defects and performed testing. The statements are validated by the main concept of ODC, i.e. a relation between defect categories and project phases, the

so called cause-effect relation visualized in the PAT, can be used to identify process problems. ODC also validates the requirement of predefined expectations of the current test model (hypotheses 5), by the purpose of binary trees, and finally the developer- and management feedback is validated by Chillarege, who emphasize the importance of developer feedback.

11.2 Risks related to the solution

The major risk with the framework is the combination of the research areas ODC and TTE, where TTE should provide supportive arguments for the mappings between test techniques and defect types in ODC. Although the result from research experiments support ODC in some aspects, they contradict in others. For instance Wood (1997) and Basili (1987) conclude that the test result depend on the program type. This makes it difficult to define expectations on test techniques, unless the program type is considered. However if it is possible to have predefined expectations on test phases or code inspection as in ODC, it should be possible to have equal expectation on test techniques on a more detailed level.

Finally it remains to see if the framework works in practice, which is up to the industry and research to evaluate. The main question for the industry is to evaluate the value of the framework in relation to the cost of developing it. The question comes back to what was stated earlier in the report, depending on if the industry wants to improve product quality or not: "*Counts of defects found during the various defect detection activities in software projects and their classification provide a basis for product quality evaluation and process improvement* [EmWi98]".

12.0 Further Work and Future Research

This chapter aim to describe which steps that remain for U/PD to do in order to implement the test framework. Further possible research areas are also identified that can improve the developed framework and testing in general.

12.1 Further work for U/PD

In order to apply the developed test framework described in the report, two phases are necessary. The first phase can be divided into three tasks and provides the base for the test process evaluation:

1. It is necessary to analyze and complement the current defect classification *Figure 15*, in order to identify all possible defect elements that can occur in the SDP, and perhaps develop more sub-categories.
2. The developed PAT has to be further complemented with test techniques and perhaps to break down some test techniques on a more detailed level (*Figure 16*). For example an inspection technique may be divided into several others as done with code inspection.
3. Parameters should be identified that can visualize the test process further, for example the logging of coverage and the programmers' expertise level. More parameters are mentioned in *Section 12.3.*

The second phase consists of the implementation of tools in order to extract and analyze project information:

1. Two checklists have to be developed for the logging of 1) occurred defects and 2) performed testing.
2. Expectations on each test technique and test phase have to be defined and logged for future evaluation. For instance which defects are expected to be detected by a certain test technique and in which test phase *Figure 15*? Further the percentage of defects detected by a test technique should be defined in relation to other test techniques, which then can be evaluated as described in *Section 8.4.1*.
3. An application based on the PAT should be developed to provide developer feedback, on which defects that could have been prevented with one or several test techniques *(Section 10.4.1*).
4. An application should be developed for manager feedback. The feedback should provide information on 1) how well the test techniques and test phases detect defects as expected, compared to other test techniques and test phases, 2) statistics of which defects that are problematic and should get extra attention, and 3) how testing is performed by the individual tester or project, i.e. are all test techniques used as expected?
5. An administration tool has to be developed in order to add, change and remove: 1) test techniques and defect categories in the PAT (*Figure 16*), and 2) defect categories and elements in the developed defect category table (*Figure 15*).

12.2 Framework value compared to the cost of implementation

The most important question when working with the industry is if the cost of implementing a solution is covered by the benefits of the solution. This question can unfortunately not be answered in this report since such an experiment has not been able to perform within the project time, therefore this remains to be studied in research.

12.3 Further logging

The developed evaluation framework is based on expectations of how testing should be performed and which defects that should be caught, i.e. the relations in the PAT. However there are several parameters that could provide great value for the evaluation of a test process. Some of these parameters that we feel are of importance, related to occurring defects, are:

- Is a detected defect due to an external product?
- How many developers participate in an inspection?
- What is the programmer's level of experience?
- How long time does an inspection take?
- How much code is covered in testing, i.e. code coverage?
- When are test cases designed, i.e. before, during or after implementation?
- How is it assured that a defect is fixed, for instance is it necessary to add test cases?

12.4 Future research on the environment's affect on testing

It has been concluded from several research experiments that the effectiveness of a test technique depends on the program type [Basili87, Wood97]. Therefore it seems reasonable to perform further research on TTE in relation to program characteristics. For example, which program characteristics results in that test technique (A) detects most defects in a certain program compared to other test techniques, where as test technique (B) detects most defects in another program type. If such relations can be established between test techniques and program characteristics, it would make it easier to define expectations on test techniques and test phases.

13.0 References

[Basili87] V. R. Basili, R. W. Selby, Comparing the Effectiveness of Software Testing Strategies, IEEE Transactions on Software Engineering, Vol. SE-13, NO. 12, December 1987.

[Beck00] K. Beck, eXtreme Programing, 2000, Addison-Wesley, ISBN: 0-201-61641-6.

[Chillarege92] R. Chillarege, S. Bhandari, Orthogonal defect classification -- a concept for in-process measurements. IEEE Transactions on Software Engineering, 18(11):943--956, November 1992.

[EmWi98] K. E. Emam, I. Wieczorek, The Repeatability of Code Defect Classifications, Proceedings of the Ninth International Symposium on Software Reliability Engineering, pp 322-333, 1998.

[Fenton] N. E. Fenton, S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, ISBN: 1-85032-275-9.

[Gilb93] T, Gilb, D, Graham, Software Inspection, 1993, Addison-Wesley, ISBN: 0-201-63181-4.

[IEEE90] Std 610. 12-1990. IEEE Standard Glossary of Software Engineering Terminology.

[Linger79] R. C. Linger, H. D. Mills, B. I. Witt, Structured Programming: Theory and Practice, Addison-Wesley Publishing Company, Inc, 1979, ISBN: 0-201-14461-1.

[Selby86] R. W. Selby. Combining software testing strategies: An empirical evaluation. In Proceedings of the Workshop on Software Testing, 15--17 July, Banff, Canada, pages 82--90. IEEE Computer Society Press, 1986.

[So00] I. Sommerville, Software Engineering, Excerpt: Verification & Validation, Addison Wesley, 2000, pp. 417-476.

[Williams00] L. Williams, R.R. Kessler, W. Cunningham, R. Jeffries, Strengthening the Case for Pair Programming, IEEE Software, July/August 2000, pp. 19-25.

[Wohlin] C. Wohlin, M. Höst, P. Runeson, and A. Wesslén, Software Reliability, Encyclopedia of Physical Sciences and Technology (third edition), Academic Press, In review, pp. 1-27.

[Wood97] M. Wood, M. Roper, A. Brooks, and J. Miller. Comparing and combining software defect detection techniques: A replicated empirical study. In 6th European Software Engineering Conference, pages 262--277. Lecture Notes in Computer Science No 1301, ed. Mehdi Jazayeri, Helmut Schauer,1997.

14.0 Appendix

14.1 A summary of the survey performed at U/PD

In the beginning of this project a survey was performed at U/PD by handing out questionnaires to eight programmers. The aim with the survey was to get a general understanding of how testing is performed at U/PD. The survey was not performed in an empirical way and it is therefore hard to draw any conclusions from the answers. Still it gives a general idea of how the programmers perform testing. Before presenting the results it is of interest to know that 50% of the developers thought that they performed enough testing.

The areas of focus are:

1. Which are the most occurring defect types in Basic Test and Function Test?
2. How and when are test cases designed?
3. Which is the programmers opinion of inspection and pair programming?
4. Which test tools are used and to what extent?

14.1.1 Conclusions drawn from the survey

Defects most often reported from Basic- and Function Test

Provided that the developers have a correct view of the failures reported from Function Test, it seems like all the developers make different defects. Each developer had different opinions about which defects that were most detected in Basic Test and Function Test. This is probably due to several parameters such as the experience of the programmer. It seems likely to believe that developers performed test differently and is focusing on different details in the code, resulting in that different types of defects are found in Basic- and Function Test.

Test Case Design

The questions covering test case design were: 1) When the test cases are planed/ designed? 2) How well are the test cases planned?

Figure 19 shows that most developers design test cases during the implementation. By doing so, there is a risk of missing test cases, which might result in failures. Optimally test cases should also be based on the requirement specification and program design.

Figure 20 shows that most developers consider both functionality and code specific details when designing test cases. It is of importance to consider both of this areas since they complement each other, which is described in *Section 6.2*.

Figure 21 shows how well the analysis of a program is made, when designing the test cases, i.e. that program domains and boundaries are identified. This part of the test case

design seems to differ a lot between the developers. This area is essential in order to perform testing thoroughly, otherwise there is a risk for reduced software reliability.

A comment that can be made based on these answers is the fact that less than 50% of the developers always perform an analysis of the program domain and boundary values, might indicate that the testing is not performed optimally. Therefore it might be questioned if in fact 50% of the developers are testing enough as they stated themselves.

	Before Design	**During Design**	**During Implementation**	**After Implementation**
Time of test case design		1	7	

FIGURE 19. Shows when the developers design their test cases, which mainly is performed during the implementation.

	Always	**Sometimes**	**Never**
Test case design in relation to Functionality	4	2	
Test case design in relation to code structure	4	1	1

FIGURE 20. The figure shows on which level test cases are designed, i.e. black-box and white-box testing.

	Always	**Sometimes**	**Never**
Identification of program domains	3	2	2
Identification of domain boundaries	1	4	2

FIGURE 21. The figure shows how well the analysis and planning is performed when designing the test cases.

Inspections

All the developers had tried inspections but one, who did not answer the questions regarding inspections, and all of them agreed on that inspections is a good test technique for detecting defects. However some developers commented on that inspections are boring to perform and sometimes the amount of code too large to inspect.

Pair programming

There were only two of the developers that had tried pair programming before and their experiences where positive. Common for all of the developers was that they all wanted to introduce pair programming into the projects.

It is interesting that some of the developers thought inspections where boring and tedious, but still in favour of pair programming, even though inspections is an integrated part of Pair Programming. However there are several types of inspections and the one performed in Pair Programming is less formal and might be more appealing to a developer. Further it is performed continuously during the project making the portions of code smaller to inspect.

Tools

U/PD has the possibility to use several types of testing tools. The ones focused on in the questionnaire are:

- *Rational PureCoverage:* Visualizing the percentage of executed code.
- *Rational Purify:* Which is detecting memory leaks at runtime
- *Rational Quantify:* Measuring the performance during execution, pinpointing bottle necks.

Even though these tools serve a purpose for the type of system developed at U/PD, only Rational Purify is used to a wide extent. Further all the developers feel the need for education on the all the tools.