LAPPEENRANTA UNIVERSITY OF TECHNOLOGY
Department of Information Technology

Software testing design and implementation in a small software company

The subject has been approved by the department council on May 14, 2003.

Supervisors:

Professor Heikki Kälviäinen

Professor Olli Martikainen

Instructor:

M .Sc. Jorma Helin

Lappeenranta 12.9.2003

Mikko Rossi
Tervahaudankatu 3 as 9
53850 Lappeenranta
Telephone: +358 40 8619416
E-mail: rossi@lut.fi

Abstract

Lappeenranta University of Technology
Department of Information Technology
Mikko Rossi
Software testing design and implementation in a small software company
Master's Thesis
2003
69 pages, 14 figures, 1 table, 1 appendix
Supervisors: Professor Heikki Kälviäinen
Professor Olli Martikainen
Keywords: testing, quality, verification, validation, software

Many software companies have started to pay more attention to improving the quality of their software product and towards that end the chosen approach taken by most of these companies is software testing. This testing should not be limited to the software product alone, but instead it should encompass the whole software development process. While validation testing is used to ascertain that the end product satisfies the requirements set for it, verification testing is used as a proactive testing, trying to prevent errors before they are implemented in the code.

The work upon which this thesis is based was done during the spring and summer of 2003 for a company, called Necsom Ltd. Necsom Ltd. is a small Finnish software company whose research and development is done at the Lappeenranta premises. This thesis gives an introduction to software testing and to different ways to organize it. In addition guidelines for creating test plans and test cases necessary for successful and efficient testing are outlined. After the theory has been gone through an example of how the software testing was implemented at Necsom Ltd. is presented. Finally conclusions are drawn from observing our testing process in practice and suggestions for further improvements are given.

Tiivistelmä

Lappeenrannan teknillinen yliopisto
Tietotekniikan osasto
Mikko Rossi

Software testing design and implementation in a small software company

Diplomityö
2003
69 sivua, 14 kuvaa, 1 taulukko, 1 liite
Tarkastajat: Professori Heikki Kälviäinen
Professori Olli Martikainen
Hakusanat: testaus, laatu, verifikaatio, validaatio, ohjelmisto
Keywords: testing, quality, verification, validation, software

Monet ohjelmistoyritykset ovat alkaneet kiinnittää yhä enemmän huomiota ohjelmistotuotteidensa laatuun. Tämä on johtanut siihen, että useimmat niistä ovat valinneet ohjelmistotestauksen välineeksi, jolla tätä laatua voidaan parantaa. Testausta ei pidä rajoittaa ainoastaan ohjelmistotuotteeseen itseensä, vaan sen tulisi kattaa koko ohjelmiston kehitysprosessi. Validaatiotestauksessa keskitytään varmistamaan, että lopputuote täyttää sille asetetut vaatimukset, kun taas verifikaatiotestausta käytetään ennaltaehkäisevänä testauksena, jolla pyritään poistamaan virheitä jo ennenkuin ne pääsevät lähdekoodiin asti.

Työ, johon tämä diplomityö perustuu, tehtiin alkukevään ja kesän aikana vuonna 2003 Necsom Oy:n toimeksiannosta. Necsom on pieni suomalainen ohjelmistoyritys, jonka tutkimus- ja kehitysyksikkö toimii Lappeenrannassa.Tässä diplomityössä tutustutaan aluksi ohjelmistotestaukseen sekä eri tapoihin sen organisoimiseksi. Tämän lisäksi annetaan yleisiä ohjeita testisuunnitelmien ja testaustapausten tekoon, joita onnistunut ja tehokas testaus edellyttää. Kun tämä teoria on käyty läpi, esitetään esimerkkinä kuinka sisäinen ohjelmistotestaus toteutettiin Necsomilla. Lopuksi esitetään johtopäätökset, joihin päädyttiin käytännön testausprosessin seuraamisen jälkeen ja annetaan jatkotoimenpide-ehdotuksia.

Preface

This thesis was written for Necsom Ltd. in Lappeenranta during the spring and summer of 2003. I want to thank both of my supervisors: Heikki Kälviäinen and Olli Martikainen as well as Jorma Helin, my principal instructor, for their guidance and opinions on my thesis. My thanks also go to my girlfriend Saija for the food and support offered during my studies and work at Necsom. I also received a lot of information and helpful tips from my brother Pekka who beat me in graduating from this school by about six months. And while I am thanking people, I probably should not forget my colleagues at Necsom Ltd.: thank you for bearing with me and making Necsom such a good place to work at!

TABLE OF CONTENTS

Terms and Abbreviations

Control flow	The sequence of execution of instructions in a program.
CVS	Concurrent Versions System
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
FTP	File Transfer Protocol
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol, Secure
IMAP	Internet Message Access Protocol
LDAP	Lightweight Directory Access Protocol
NTP	Network Time Protocol
PAP	Push Access Protocol
POP	Post Office Protocol
RADIUS	Remote Authentication Dial In User Service
SMTP	Simple Mail Transfer Protocol
SQAP	Software Quality Assurance Plan
Stub	A dummy program module used during the testing of a program.
Telnet	The network terminal protocol
Test driver	If a function or procedure cannot be called directly, a test driver is used to make these calls. Used during unit and integration testing.
Testware	Collection of test cases, test drivers, stubs, scripts, and files
WAP	Wireless Application Protocol

1 Introduction

1.1 Background

Customers are not anymore looking for only new features in the software they are buying but instead they are starting to demand more reliable and easy to use applications. Because of this trend more and more companies are starting to shift some of their resources towards improving the quality of their software and the software development process as a whole. Software testing is one of the available tools for striving towards these goals. It is a complex process that requires a structured and methodical approach to it for the testing to be both effective and successful. Software testing is often used first and foremost to ascertain that the end product meets the requirements set to it, but there is much more to it than executing the software product and observing the results.

1.2 Objectives and restrictions

The objective of this thesis is to study how the software testing should be organized, planned and implemented in general. Also an overview of software testing is introduced: general strategies for testing and how to divide the test process into more manageable levels. In addition to the above some guidelines are provided for designing and prioritizing test cases. The ultimate goal was to decide how the software testing should be organized and implemented at a company called Necsom Ltd.

It is assumed in this thesis that at least a high level requirement specification is available for use as a base when planning and designing testing. This thesis does not cover testing techniques in any great detail, only a short description of some testing techniques is provided within this work. This thesis will not cover object-oriented testing, the costs of testing or test coverage. Also formal ways to calculate the priority of a test case, or the probability and severity of a potential problem are left out of the scope of this thesis and only a few general ideas are presented how these values can be used.

1.3 Structure of Thesis

This thesis contains five chapters in addition to this introduction. Chapter 2 gives the reader a general overview of the testing process and how it can be divided into smaller elements. Chapter 3 covers the organization of testing and the different roles associated with it. In chapter 4 the planning of verification and validation testing is introduced. Also guidelines for test case design and prioritization are explained in this chapter. Chapter 5 gives an example how testing has been organized and implemented at Necsom. Chapter 6 is reserved for conclusions drawn by this thesis.

2 Overview of software testing

There are many different definitions of testing introduced in texts covering software testing. This chapter introduces some ways which can be used to realize these definitions. Some of these definitions are:

“The aim of software testing is to guarantee, as far as possible, that the software is correct.” /1/

“Testing involves operation of a system or application under controlled conditions and evaluating the results” /2/

2.1 Software development models

Software development models are used for describing the work flow associated with software development process. Two of the most commonly used development models are the modified waterfall model and the V-model.

2.1.1 Modified waterfall model

The waterfall model (Figure 1) is one of the oldest ways to model the software development process and life cycle. The development is divided into a number of steps that are gone through in a descending order starting from the top. At the end of each step it is evaluated if all the requirements for the current step are met and if the next step can be started. If the requirements are not met the development should continue on the current step. In some cases it may be necessary to go back one or more steps and start descending all over again if changes are necessary to accommodate late requests by the customer or to correct a fundamental error introduced in the design of the software. The original waterfall model did not have any feedback mechanism built into it and this mechanism has been added in the modified waterfall model.

The starting point in the waterfall model for the software development cycle is the idea of the software that will be developed. After it is established that there is a demand for this product the requirements that this software must fulfill have to be specified. When all the requirements are defined the design phase can be started. The design phase is focused on the data requirements, the software construction and the interface design and coding. When the design phase is completed the implementation phase starts and the actual application will be programmed. After all these steps have been gone through the software still needs to be tested to verify that it meets all the requirements set for it. After the software has passed all the tests performed the product is delivered to the customer and it will be maintained by fixing any defects that have gone unnoticed and possibly by introducing minor improvements.

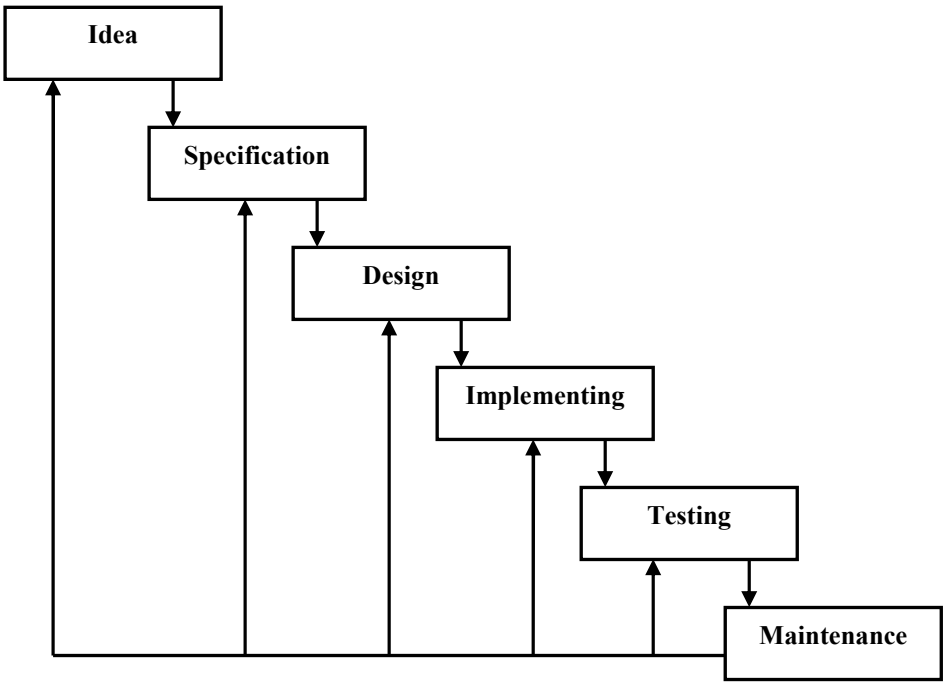

Figure 1: Modified waterfall model (the picture has been modified) /3/.

There are some well known problems when employing the waterfall model in software development. First of all the requirements change frequently during the development and if they change after the specification phase, it is expensive to restart from the specification step, especially later during the development cycle. Even if the possibility of

moving back through the steps is introduced in the waterfall model, most projects tend to employ the model in a linear fashion and try to avoid backtracking at all costs. Another problem is that the errors are discovered late during the cycle and it is an established fact that the expense of correcting an error is progressively higher the later it is discovered.

Even with these problems this model is widely used and the steps it defines are simple and essential to any software development process. Many variations and even completely new models have been developed with the waterfall model as the basis for them.

2.1.2 V-model

The V-model /4/, shown in Figure 2, is derived from the waterfall model, with special attention given to the testing of the product and the different documents generated during the development cycle. The V-model clearly connects the different phases during the development cycle (on the left) into the phases that are used to validate their implementation (on the right).

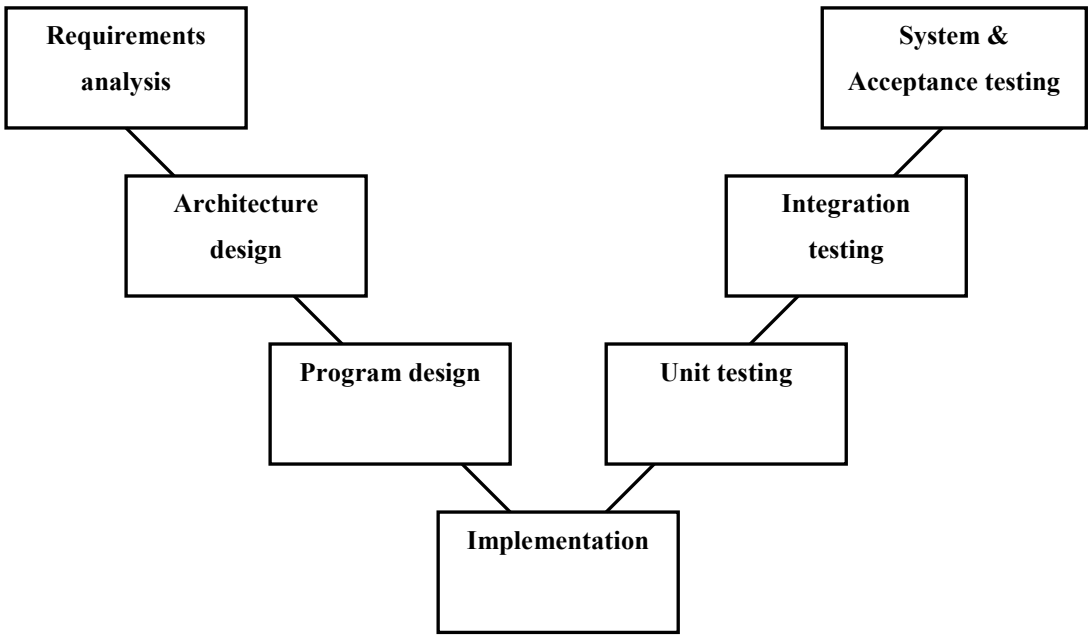

Figure 2: V-model (The picture has been modified).

In the requirements analysis phase the requirements for the product are gathered from the customers and identified by the marketing personnel and then these requirements are evaluated to determine which of them should be implemented. These requirements are entered to the requirements specification for the product that will be developed. In order to avoid introducing new requirements at a later stage that will increase the cost and time required to develop the software, it is highly recommended to involve the testing personnel already at this stage since they will most likely have a few requirements for the software that will make it more testable, for example concerning different aspects of logging.

During the architecture design phase the high level technical specifications concerning the architecture and overall implementation of the product are designed. The resulting technical specification needs to be reviewed by the development team and the development manager responsible for the implementation of the software.

A detailed design document for each individual unit is generated during the program design phase. Each of the produced design documents is reviewed within the development team.

V-model's strength is that it gives the testing of the product an equal weight when compared to the designing and implementation of the software. However the V-model shares the same problem as the waterfall model: it is assumed that once the requirements and designs are decided upon, they will not change afterwards.

2.2 Testing methods and strategies

Traditionally testing strategies (or test design methods) have been divided into two fundamental strategies: black-box and white-box testing. Testing strategies have also been divided into verification and validation testing depending on what it is exactly that is under examination. Of these two strategies the validation testing is further divided in Edward Kit's book into requirements-based tests, function-based tests and internals-based

tests which differ in what information is used for designing the tests. /5/ This division is presented in Figure 3.

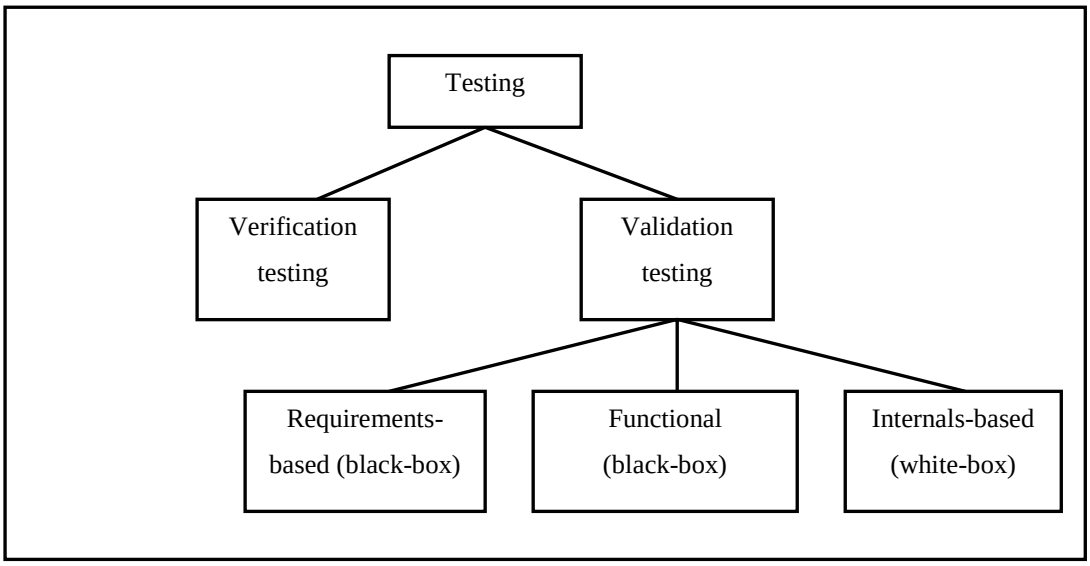

Figure 3: Common testing strategies.

2.2.1 Black-box testing

Black-box testing is a fundamental testing strategy, where the test cases are designed without any knowledge of the internal workings of the software under testing. This means that the tester does not (and should not) need to have any access to the source code when designing test cases. When designing black-box tests, the tester should learn about the user, his expectations and needs, different configurations the software will run on, the other applications that this software will interact with and the data this software has to manage. /6/

Black-box testing has both pros and cons to it. Black-box testing is more effective for large units than white-box testing, tests are done from the end-user point of view and test cases may be designed as soon as the requirements have been defined. Problems may be caused by the facts that certain paths through the application will never be tested if only

black-box testing is used and test case design will be much harder if the requirements are not well documented.

2.2.2 White-box testing

In white-box testing the tester has access to the source code and internal workings of the tested component. It may be thought that the tester can see inside the box that he is testing and can use that knowledge to locate weaknesses and potentially error prone areas in the component. In some cases it possible to find errors that would not be found if the tester resorted only to black-box testing, but there is always the possibility that when using only white-box testing it is checked that the code works as it was written as opposed to checking that logic of the code is correct.

2.2.3 Verification testing

IEEE/ANSI defines verification testing as "The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase." This is somewhat vague although verification testing is generally thought of as a proactive type of testing. Verification is, to a large extent, the Quality Control activities that are done throughout the lifecycle that help to assure that interim product deliverables meet their initial specifications. /7/ Verification is done for the different documents produced during the software development life cycle, such as requirements specification or the system test plan. Basically verification testing answers the question: “Are we building the product right?”

2.2.4 Validation testing

The IEEE/ANSI definition of validation is "The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements." /7/ Validation works with the code when the project has

progressed far enough to have some implementation that can be executed. In essence validation testing answers the question: “Are we building the right product?”

Validation testing can further be divided into requirements-based, functional tests and internals-based tests. Requirements based tests are designed based on the requirements specification for the software. Typically the fundamental testing strategy utilized in requirements-based tests is the black-box strategy.

Black-box testing should also be used for functional testing, but the basis for the test cases is taken from the functional requirements for the application. Unlike in the internals-based tests it is not necessary to have any access to the source code itself.

Internals-based (or structural) tests employ the white-box testing strategy and therefore require a certain degree of familiarity with the code, but the test cases may be still based at least partly in the functional requirements specification.

2.3 Testing phases

The testing process is often divided into different levels or phases to make the whole process more manageable. These levels are designed to build upon each other, making the discovery of errors easier by first testing the application in smaller and less complex pieces. It is quite common to partition the testing process into the following phases:

- Unit testing
- Integration testing
- System testing
- Acceptance testing
- Regression testing

2.3.1 Unit testing

The purpose of unit testing is to test the smallest possible parts of the software separately. The unit is often defined as the smallest component of the software that can be compiled. Because unit tests are by the very definition itself conducted upon separate entities of the software without the help of other components, unit testing requires the use of drivers and stubs. Driver is a test module that executes the unit that is being tested and stubs simulate the missing components that the unit would interact with when the application is completely implemented.

Unit testing is most often associated with white-box testing, but it is also possible to use black-box testing as well if there are functional requirements for the unit under testing. Whatever the strategy chosen, unit testing is the first opportunity to exercise the code and locate errors within it. Because there are no other components complicating things, it is often easier to find errors, evaluate the underlying reasons and necessary corrective actions for them.

2.3.2 Integration testing

The process of combining the units to form bigger bodies of code is called software integration and the process of testing these bodies is called integration testing. Usually the integrated units should have gone through unit testing before the integration testing so that the integration testing can concentrate on finding errors in the interfaces and communication between the units. The idea behind integration testing is that when a problem is found during the testing, it should at least theoretically be easier to find the root causes for these problems because the most recently added units are likely the sources of the problems. /4/ Integration testing also makes it possible to test parts of the software before all of the implementation is ready.

Louise Tamres /4/ has chosen to divide the integration testing into four different strategies:

- Top-down strategy can proceed in either depth-first or breadth-first manner. For depth-first integration each module is tested in increasing detail, replacing more and more levels of detail with actual code rather than stubs. Alternatively breadth-first would proceed by refining all the modules at the same level of control throughout the application /8/. This method requires stubs to simulate the lower level components.
- Bottom-up: first unit test the lower level component and then have that component be called upon by a higher level component. This method requires drivers to run the components being tested.
- Sandwich combines the top-down and bottom-up methods.
- Big-bang assumes that all the units are integrated at once and then tested together for the first time. This method does not require any drivers or stubs, but it defeats one of the ideas behind integration testing: it will be somewhat more difficult to locate the root causes for encountered problems.

2.3.3 System testing

After all the units that form the system are fully implemented and integrated the system testing can begin. System testing is used to test the whole product from a user's point of view and the test cases associated with it are derived from the requirements set for the product. During this phase of testing the testing environment should be as near the environment the software will be deployed upon as possible.

Testing techniques especially associated with system testing are compatibility and interoperability testing, stress testing, installation testing, fault recovery testing, usability testing, reliability testing and security testing.

2.3.4 Acceptance testing

Acceptance testing is conducted to determine whether or not a system satisfies the user's requirements and to enable the customer to determine whether or not to accept the system. It is surprising how often there are misunderstandings on what some particular requirement actually means and that is why the acceptance testing should be done by both the user's and software company's representatives with assistance provided by the testing team. The representative of the company will test the software against the final requirements set for the software whereas the user's representative tests that the software works as the users want it to work.

Acceptance testing is preceded by the system testing, which should be successfully completed beforehand. Test cases for acceptance testing usually are a subset of test cases taken from system testing. This phase of testing is quite often conducted in the real deployment environment provided by the customer and it is the final testing conducted before delivering the product.

2.3.5 Regression testing

Some of the components that make up the system always change at some point during the system's life cycle. This happens when error fixes and improvements are introduced to the code. If the component has already been tested before these changes, the previous functionality should be tested all over again to make sure nothing was broken by these changes. This is called regression testing.

A subset of previously executed test cases forms the set of test cases that are used during the regression testing. Testers should take care to keep regression test cases up to date, because some of the changes may invalidate old test cases. They should also bear in mind that if new features were implemented, the regression testing alone is not enough and they should also create and execute new test cases for these new features.

2.4 Testing techniques

"A test technique is a way of creating or executing tests" /9/. There exists a great number of different testing techniques and some of them are shortly described here.

2.4.1 Reviews and walkthroughs

Reviews and walkthroughs are done to both the documents produced by the development and testing team and the code of the application that will be tested. Review is a more formal method of going through the product being inspected than the walkthrough. The main difference between these two methods is the reviews require at least some amount of preparation by all the participants whereas walkthroughs require preparation only by the presenter of the document or code.

2.4.2 Equivalence partitioning

This testing technique is based upon the assumption that the inputs of the application can be divided into separate classes and that all the members of these classes can be trusted to work in similar manner to each other. If we accept these assumptions to be correct then only one member of each class needs to be tested. The difficult part with this technique is to determine how to divide the application into different classes. /10/

2.4.3 Boundary analysis

Boundary analysis is a test technique in which test data is chosen to lie among boundaries or extremes of input domain (or output range) classes, data structures or procedure parameters. Boundary value test cases often include the minimum and maximum in-range values and the out-of-range values just beyond these values.

2.4.4 Path testing

This is a testing technique where different paths through the program are selected based upon the program's control flows and it is the cornerstone of testing. /11/ This technique is especially useful when unit testing a module.

2.4.5 Transaction-flow testing

A transaction is a unit of work seen from a system user's point of view and transaction flows are introduced as a representation of a system's processing. Transaction flows and transaction-flow testing are to the independent system tester what control flows and path testing are to the programmer. /11/

2.4.6 State transition analysis

In state transition analysis the program is modeled as a finite-state machine and this model is then used to test the transition between the different states to find out problems and inconsistencies in the function of the program.

2.4.7 Compatibility and interoperability testing

This sort of testing is used to test that the software works as it is supposed to work with other running applications in the system and that it does not hamper them in any way. Interoperability testing verifies that the software being tested interacts correctly with other applications it is integrated with. /10/

2.4.8 Smoke testing

This is usually done during regression testing to prove that a new build is not worth testing. Smoke tests are often automated and standardized from one build to the next /6/. These tests are often simple and test the basic functionality of the software, the rationale being that if these tests do not work this build should not be tested any further before the problems have been fixed.

2.4.9 Error guessing

Error guessing should be used only as a complementary testing technique. In it an individual tester tries to locate the errors in the program based on his expertise, previous experience with similar programs and intuition.

2.4.10 Fault recovery testing

This testing technique is used to ascertain that the application can recover from error situations like system crash or unexpected exceptions during code execution and can resume a legal state, for example by automatically restoring the situation before the action that resulted in the exception.

2.4.11 Installation testing

When installing the software under testing, the installer must work properly. This means that everything needed for the use of the program should be placed to the correct locations (executables, required libraries, configuration files, possible documentation) and initial configuration is done. If there is an installation document for the software, it should be checked that it is correct and covers all the necessary requirements and steps for installation.

2.4.12 Documentation testing

All the documentation concerning the product should be tested by reviewing and using the software as described in them. The documentation to be tested is not limited only to user documentation: it extends to the requirement specifications for the program, design documents, test plans, test cases, code commenting and interface specifications.

2.4.13 Performance testing

Performance testing is used to test how the developed program and its operating environment respond to various levels of load placed upon the program (e.g. many simultaneous users, a query to a very large database or multiple concurrent processes). Interesting metrics are for example the processor load, network traffic, consumed memory and needed disk space.

2.4.14 Reliability testing

The system is presented with a large number of ‘typical’ inputs and its response to these inputs is observed. The reliability of the system is based on the number of incorrect outputs which are generated in response to correct inputs /12/.

2.4.15 Stress testing

Stress testing is subjecting a system to an unreasonable load while denying it the resources (e.g., RAM, disc, mips, interrupts, etc.) needed to process that load. The idea is to stress a system to the breaking point in order to find bugs that will make that break potentially harmful. The system is not expected to process the overload without adequate resources, but to behave (e.g., fail) in a decent manner (e.g., not corrupting or losing data) /13/.

2.4.16 Security testing

Security testing attempts to violate built in security controls. It ensures that the protection mechanisms implemented in the software application secure it from abnormal penetration. Examples for this are as simple as making sure the third failed logon attempt locks the user out or ensuring that all transmitted data is encrypted. /14/

2.4.17 Usability testing

Usability Testing is a process that measures how well a web site or software application allows its users to navigate, find valuable information quickly, and complete business transactions efficiently. Throughout a testing session a team of Usability Analysts carefully watch the user's actions and listens for feedback related to the design or interface /15/. This technique requires a special usability laboratory often equipped with video cameras and other hardware and software to track the actions taken by normal users doing the testing.

2.4.18 Automation of testing

The automation of testing comes in very handy when associated with regression testing phase, because at that time the tests have already been executed once, the expected result is already known and the results can be compared against it. Since these tools can run unattended for extended periods of time this reduces the time required for testing and may improve the quality of testing by lessening the workload placed upon the test staff.

Stress testing and performance testing are examples of testing techniques used with automated testing. However it must be realized that the automation of tests is not suitable for all testing tasks and should never be used as the only means of testing an application. Furthermore the automation of testing is very similar to a software development process: a tester has to design and implement the test automation using the tools available to him.

2.5 Reporting

Reporting consists of problem reporting and the different reports delivered to the project management. Most of the reporting is done via problem reports, but in addition to these at least a test report should be generated when the testing is finished. Other reports that are usually generated are reports on testing statistics and progress reports, both of which should be generated periodically during the test execution.

2.5.1 Problem reporting

The problem reporting consists of the problem reporting system and the problem reports inserted into it. The problem reporting system is often the primary way of communication between testers and developers. From the problem reports generated by the testers (and possibly users of the software if the problem reporting system is available to them) the developers get a clear picture of the issues discovered with the system they have implemented and find a starting point from which to start analyzing the errors and debugging the code.

The problem reporting system should not only store the problem reports but it should also support severities and priorities for the reported problems, the assignment of people responsible for analyzing and correcting those problems and manage the current status of each problem report. It should also keep a record of the date the problem report was submitted, changes made to the problem report and their date as well as allow the attachment of relevant files like screenshots or log files to the submitted problem reports.

The problem reports generated by the testers must have a unique identity number, severity of the problem, the name of the person who reported the problem and of course the description of the problem.

2.5.2 Test report

A test report is generated when the corresponding test phase is finished and it is delivered to the project management. This test report should contain the following items:

- Reference to the corresponding test plan.
- Deviations from the original plan.
- Status of all the entry/exit criteria specified for the test phase.
- Statistics on test cases.
    - Number of executed test cases.
    - Passed test cases.
    - Failed test cases.
    - Test cases that could not be executed.
- Statistics on defects.
    - Total number of discovered defects.
    - Amount and state of the defects of different severities.

3 Organizing testing

Much of this chapter is based on books by John Watkin: Testing IT, An Off-the-Shelf Software Testing Process /10/ and Edward Kit: Software testing in the real world – improving the process /5/. I found both of these books to be well written and easy to read.

3.1 Organization of testing

Organizations exist because there is a need for people as a group to behave predictably and reliably. To achieve results, people need to cooperate with each other. Organizations are created to support and coerce the activities people engage in. Good organizations minimize the inevitable conflict between the needs of the organization and the needs of the individual. /5/

Testing cannot be isolated from all the other activities of any organization, so thought should be given to which different departments and people the testing process needs to be integrated to. It is also essential for software testing to be supported by the upper management where the decisions concerning the budget and allocation of resources are made. One good and simple way to gain that support and influence the allocation of resources is to tell people what you are doing and how you are going to do it. These factors should be taken into account when planning the organization of the testing program. Other factors that should be considered as well are the size and complexity of the business, the geographic location of offices and the balance of software developed by the business itself as opposed to software developed by a third party.

3.1.1 Testing is each unit's responsibility

In this approach to organizing testing the development team members are assigned to test each other's code, so that nobody tests the code they themselves wrote. The obvious problem with this approach is that too much is expected from all the team members:

everyone should have the abilities and skills of both the developer and the tester. And in a case where time is running out, most everyone chooses to complete their coding tasks first and do only very minimal testing knowing that if there is no code there is nothing to be tested anyway. This approach is shown as the case 1 in Figure 4.

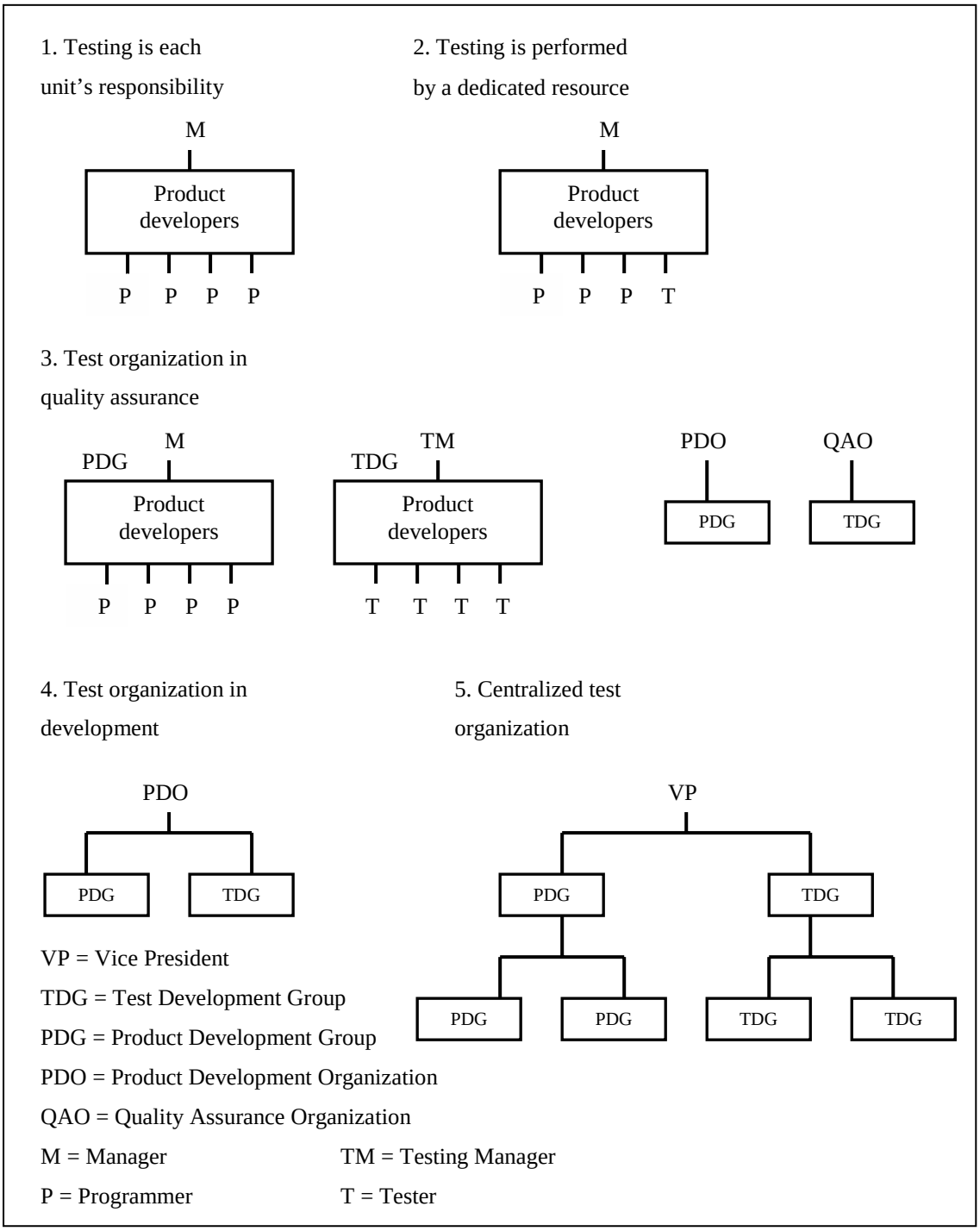

Figure 4: Different approaches to organize testing. /5/

3.1.2 Testing is performed by a dedicated resource

When testing is organized this way, there is at least one person who is solely doing the testing of the product and the programmers can concentrate on coding. However the original problem with one person doing too many things is now pushed upwards: the manager of the product development group has to manage both the programmers and the testers whose tasks and needs are very different from each other's. This is case 2 in Figure 4.

3.1.3 The test organization in quality assurance

If there is already a quality assurance department, then an easy solution would be to place the test organization as a part of it. This puts a lot of responsibility in the hands of the manager of quality assurance, since he is the one managing the testing manager as well as auditing the development process. The quality assurance manager may not understand software testing and may also introduce team work problems between the product development and test development groups. This approach might also result in the product development group feeling that they no longer have the necessary resources to produce a quality product. The good thing with this approach is that the testing manager is not over burdened with different responsibilities any more. This approach is shown as case 3 in Figure 4.

3.1.4 The test organization in development

This organizing approach is presented as case 4 in Figure 4. The testing organization may also be made a part of the development organization and thereby solve the problems with the teamwork and production of quality software. While not necessarily solving the teamwork problem completely, it at least helps with it by bringing the product development and test development groups closer to each other. However the problem

with placing a great deal of different and sometimes even conflicting responsibilities is transferred from the quality manager to the product development manager.

3.1.5 Centralized test organization

The objective of creating a centralized test organization is to solve the senior management problem, where a lot of work is placed upon one single manager. Now there is one senior testing manager who manages other test managers. Of course this would still seem only to push the problem upwards to the lap of the vice president, but at this point the management decisions made by the vice president are so high level, that intimate knowledge of testing is not as important as it is for the testing managers. Naturally the vice president has to be able to trust his senior product development and test development managers to make this approach really work. Centralized test organization is displayed as case 5 in Figure 4.

3.2 Roles in testing

The allocation of roles is very much dependent on the size of the business where testing is performed, so the roles described below are not limited only to the tasks detailed for them. Indeed the roles may even be combined, so that for example the testing manager may also be a test team leader for the same project. The different roles in test development group are depicted in Figure 5.

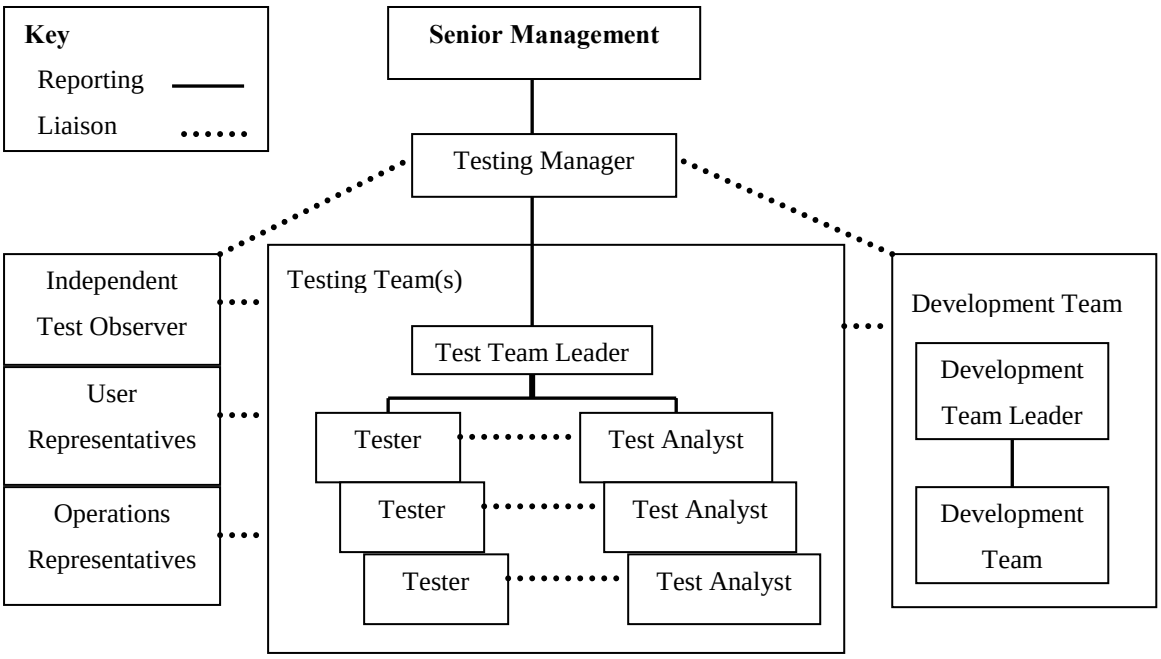

Figure 5: Organization of the Test Development Group /10/

3.2.1 Testing manager

Managing testing requires a lot from the testing manager: the manager must be able to understand and evaluate the software test process as a whole: the standards, policies, tools and training associated with it. Based on this evaluation, the manager strives to improve the quality of the process. He or she must be able to maintain a strong, independent and formal test organization and have the ability to lead, communicate and support it.

The Testing Manager has the authority to administer the organizational aspects of the testing process on a day-to-day basis and is responsible for ensuring that the individual testing projects produce the required products to the requires standard of quality and within the specified constraints of time, resources, and costs. /10/

Testing manager may additionally be responsible for liaising with development teams, so that the unit and integration test phases are conducted properly. Testing manager also liaises with the independent test observer to keep an eye on how the testing is proceeding from the view point of the observer. The test plans generated by test team leader are

reviewed by the manager to make sure they cover all the necessary issues. Other tasks for the testing manager include things like test resource planning and allocation, task identification and effort estimation, task and project scheduling, project tracking and oversight.

3.2.2 Test team leader

The main responsibility of a test team leader is to allocate tasks to testers and test analysts in his test team. The test team leader has the authority to run a testing project and has the responsibility to generate the test plan and test specification documents. The test plan is used for management control during the whole testing process and should be reviewed by the testing manager. Test specification document is a more detailed description of testing, the resources needed and the testing environment. /10/

The test team leader will also review the test case documents produced by the test analysts and liaise with the independent observer and development team leader during the testing. When testing progresses to acceptance testing the test team leader also needs to liaise with user and operations representatives to obtain users to do user and operations acceptance testing. At the end of testing, the test team leader files all the documentation and materials provided by testers and test analysts, so the tests can be executed exactly the same way as before and the results can then be compared.

3.2.3 Test analyst

The responsibilities of the test analyst include the analyzing of the requirements specification and the design and implementation of one or more test suites and the test cases associated with them based on that analysis. Other responsibilities for the test analyst are the prioritization of test cases and perhaps assisting the test team leader with the generation of the test plan document. The test analyst liaises with testers in the test team to brief them on their tasks before testing. When the testing process is completed,

the test analyst is also responsible for the backup and archival of all testing documentation and materials. This material will be delivered to the test team leader. /10/

Test analyst might develop scripts and small testing programs to help in testing and even facilitate the automation of testing. In some cases the test analyst may also be called to troubleshoot and respond to user problems.

3.2.4 Tester

The primary responsibility of the tester is to execute the test cases designed by a test analyst and to evaluate and document the outcome of those tests based on the expected results detailed in the test cases as well as his or her personal experience.

Before the testing can commence, the tester will setup and initialize the test environment plus any software required to support the test. During the testing, the tester will document the observed results of test cases and works closely with the independent test observer to ascertain test cases are followed correctly and precisely. When testing is finished, it is the responsibility of the tester to archive all the test results, test data, simulator or test harness software and the specification of hardware used in testing and deliver them to the test team leader, so the test environment and tools can be reassembled to do the testing again should the need ever arise. /10/

3.2.5 Observer

The observer has to be independent from the development of the software under testing and should not be in any way involved in its testing process. This way he may provide independent, objective and trustworthy verification that the testing proceeds correctly and according to the plans made for the testing process of the organization in question.

During the testing the observer needs to be present to follow that the tester executes the tests as specified in the test cases and that he or she interprets the results of testing

accurately. If the expected results differ from those received from the execution of the test, the tester documents these conflicts correctly. The observer may also be called upon to review some documents produced during the testing, like the test specification document. The observer liaises with the testing manager and reports to him his or her observations on testing. Additionally the observer needs to communicate with the testing managers and testers to schedule his or her presence during the testing. /10/

4 Test planning and test case design

For the testing to be effective it must be properly planned and this requires a methodical approach to it. An overview of testing tasks is as follows: /5/

1. Master test planning task
2. Verification testing tasks (per activity)
    - Planning
    - Execution
3. Validation testing tasks (per activity)
    - Planning
    - Testware development
    - Test execution
    - Testware maintenance

4.1 Master test planning

The purpose of master test planning is to get an overall picture of the testing process in a very high level document: master schedule, master resource usage, master life cycle and quality assurance issues, overall testing strategy and what kind of testing will we be doing. /5/

There are two main documents that should be produced as an outcome of master test planning: the software quality assurance plan and the software verification and validation plan. Other documents may be generated as necessary.

The software quality assurance plan (SQAP) is the highest level testing related document and it requires a software verification and validation plan. The purpose of SQAP is to show the user, the developer and the public the measures taken to ensure the quality in

the software. The SQAP is prepared by a software quality assurance group or an appropriate representative body. /5/ The SQAP is displayed in Figure 6.

The software verification and validation plan is intended to describe the verification and validation activities that will be applied during the project. The outline of this plan is shown in Figure 7.

Software quality assurance plan (outline)

(IEEE/ANSI, 1989 [std 730-1989])

- Purpose
- Referenced documents
- Management
- Documentation
- Standards, practices, conventions, and metrics
- Life-cycle verification and validation
- Reviews and audits
- Test
- Problem reporting and corrective action
- Tools, techniques, and methodologies
- Code control
- Media control
- Supplier control
- Records collection, maintenance, and retention
- Training
- Risk management

Figure 6: The outline of the software quality assurance plan. /5/

Software verification and validation plan (outline)

(IEEE/ANSI, 1989 [std 1012-1989])

- Purpose
- Referenced documents
- Definitions
- Verification and validation overview
    - Organization, master schedule, resources summary, responsibilities, tools, techniques, and methodologies
- Life-cycle verification and validation
    - Tasks, inputs, and outputs per life-cycle phase
- Software verification and validation reporting
    - Describes content, format and timing of all verification and validation reports
- Verification and validation administrative procedures
    - Policies, procedures, standards, practices, conventions

Figure 7: The outline of the software verification and validation plan. /5/

4.2 Planning verification testing

According to Kit /5/ here is a list of the different things for consideration when planning verification for a software project:

1. Activities
    - Requirements verification
    - Functional design verification
    - Internal design verification
    - Code verification
2. Verification tasks (per activity) include:
    - Planning
    - Execution

A separate verification plan should be prepared for each activity and it needs to describe how the verification will be performed, areas that will and will not be verified as well as

the risks and priorities for those areas. And as any good plan, the verification plan must also cover the resources, schedule and responsibilities of the people involved in verification testing.

4.3 Planning validation testing

A test plan states what the items to be tested are, at what level they will be tested, what sequence they are to be tested in, how the test strategy will be applied to the testing of each item, and describes the testing environment. A test plan may be project wide or, may in fact be a hierarchy of plans relating to the various levels of specification and testing /16/.

As a rule all the test plans (except unit test plan, which may be kept very simple) should contain at least the following items of interest:

- Test objectives.
- Test constraints.
- Risks.
- Dependencies.
- Entry and exit criteria.
- Roles and responsibilities.
- Schedule.
- Training requirements.
- Reporting.

4.3.1 Unit test planning

As unit testing is usually done by the development team itself and in the development environment which lessens the possibility of resource contention, the unit test plans do not need to be as formal and detailed as the plans for other testing phases. However the

time to execute unit tests, fixing the possible problems found and retesting has to be factored into the development schedule.

The testing strategy most often used in unit testing is white box testing, which is why the same programmers who have coded the units being tested should design the unit test cases. When ready, these test cases are delivered to the development team leader, who in turn generates a unit test plan stating what resources are needed and how much time is available for unit testing. The execution of unit test cases is also done by the same people who coded the units originally.

Testing techniques usually utilized with unit testing include both functional testing: path testing, equivalence partitioning, boundary analysis, state transition testing, code reviewing and code walkthroughs, and nonfunctional testing: performance testing, reliability testing and stress testing.

4.3.2 Integration test planning

Integration testing is closely related to unit testing and sometimes it may even be hard to separate the two from each other. Therefore creating the integration test plan is occasionally chartered to development team leader. Because integration testing is commonly done with development team's resources, the time and resources required by the integration test phase has to be included in the overall development plan to reduce resource contention. If the integration testing is done in the actual environment where the application will be used by the customers, the development team leader will have to plan with the systems administrator the installation and configuration of both hardware and software necessary for testing.

The general testing strategy used for integration testing is black box testing and appropriate testing techniques are for example: path testing, equivalence partitioning, boundary analysis, fault recovery testing, state transition testing, reliability testing and

performance testing, remembering that the emphasis on this testing phase is placed upon the interfaces and the interoperation of different modules.

4.3.3 System test planning

When the system test planning is started, the requirements specification, the design documents and the user documentation for the software should be ready and available. The system testing is planned and conducted by the testing team and the testing is conducted in the live environment and preferably with live real life data as input for the testing. The individual test cases are designed by a test analyst based upon requirements set for the software and these test cases are executed by a tester. It is the test team leader's task to liaise with the development team leader to monitor the schedule of the software development and to invite him or her to observe the system testing with an independent observer. The test team leader must also keep the systems administrator up to date with the requirements and progress of testing, so that everything is ready when the testing should be started and the systems administrator knows when the testing should be concluded.

The overall testing strategy for system testing is black box testing and the techniques associated with system testing are transaction-flow testing, installation testing, security testing, reliability testing, fault recovery testing, compatibility and interoperability testing, documentation testing and performance testing to mention a few. The use of error guessing may be considered as a complementary testing strategy if a suitable highly skilled individual is available to perform testing according to it. Also the automation of testing should be given some thought, especially so when executing the performance and stress testing.

4.3.4 Acceptance test planning

The acceptance testing should be planned in cooperation with the customer if applicable. The test team leader oversees the testing process to make sure enough tests are executed

to ascertain the proper working and quality of the software and to invite the user and operations representatives to conduct the tests. Because acceptance testing should be conducted in the real working environment using live data as input for the software, the test team leader will have to liaise with the customers or the systems administration to reserve and prepare the testing environment. In addition he will also have to follow the progress of the development team by liaising with the development team leader, especially if there were errors found during the last stages of system testing which have been corrected for acceptance testing.

The acceptance testing normally consists of a limited number of test cases chosen from system testing because the execution of all the test cases would be very time consuming. It also is the test team leader's job to work with the user representatives and choose sufficient test cases to be executed. Testers are there to set up the testing environment and help the representatives to execute the tests and check the results. These tests should be overseen by an independent observer to prevent the testers from guiding the representatives to take certain courses of action when using the software or accepting the results of a test which are subject to doubt.

The black box testing strategy is used when going through acceptance testing and testing techniques are essentially the same as during system testing with the possible addition of usability testing. If usability testing is employed a test analyst should design the necessary test cases and the team leader should acquire the necessary equipment to execute and document the testing.

4.3.5 Regression test planning

Because of the nature of regression testing, the person responsible for designing the regression test plan is the person who designed the test plan of the corresponding test phase. The necessary human resources will be drawn from the testing team and regression testing will be performed in either a dedicated testing environment or the actual working environment for the software, depending on which is more suitable for the

regression test cases. The test team leader must be in contact with the systems administrator in case the environment needs be reserved or created.

The testing strategy and testing techniques again depend on the changes made to the software and which phase of the testing is being conducted: white box testing for unit tests and black box testing for other testing phases.

4.4 Test case design

A test case is a document that describes the environment for test execution, how the test should be executed and what the expected results for that test are. A good test case should strive to be as simple and reusable as reasonable. The test case should have at least the following properties:

- Unique identification number.
- Test environment.
- Preparations necessary to execute the test.
- Instructions for test execution.
- Expected results of the test.
- Space reserved for recording the results of the test.

Every test case needs a unique identification to make it possible to trace the test cases to bug reports and other documentation related to the testing process. Every test case should also have a detailed description of the environment necessary for the test case: if the test case needs to be rerun after the original test environment has been dismantled, it is essential to know the necessary hardware and software for the execution of the test case.

Each test case should be independent from the other test cases so that these test cases can be executed separately. If a test case requires that some other test cases are executed before it, this requirement has to be informed in the section detailing necessary preparations for the case. Also all the other necessary preliminary activities have to be documented here. This adds to the reusability of the test case.

Instructions for executing the test case itself are a clear requirement for the test case document, but these instructions alone are not enough: the expected results must also be included in the test case. The presence of the expected results allows comparing them to the actual results of the executed test case. Based on this comparison the tester can validate if the software complies with the requirements set to it.

4.4.1 Prioritizing test cases

Giving a priority to a test case is a simple tool to decide which test cases can be left out from a set of test cases or to decide which test cases should be executed first when it looks like the time is running short for the testing. In her book Louise Tamres /4/ lists four reasons why the test cases may have to be reduced:

- Imminent ship date.
- Impossibly large number of test cases.
- Limited staffing resources.
- Limited access to test equipment.

The priority given to a test case is a rational and clear criterion used to choose which test cases to omit and which to execute. The priority can be determined by considering the questions: “what features must be tested?” and “what are the consequences if some features are not tested?”. After the test cases have been selected, the project management and the test team must be confident that the end product will be adequately tested.

The priority range for the test cases can be whatever feels suitable at the time it is decided as long as the range and criteria for the priority stay constant for all the test cases. When all the test cases are rated with the same criteria, they can be evaluated for inclusion in the tests to be executed. If the first round of evaluations results in too many or too few test cases the prioritization can be done again by adjusting the priority levels of the last round of evaluation. For example if the first iteration round had used a priority range from one to three, the ratings one and two can be further divided down, so that we would end up with priority levels 1a, 2a, 3a, 4a and 5a. /4/ These evaluation rounds can be

continued as long as necessary to identify the necessary test cases. It is of course possible that the necessary number of test cases simply can not be executed in the time available for testing no matter how many prioritizing rounds are done. In such a case the project management must find other solutions to the problem: maybe more personnel can be allocated to testing, the shipping date has to be moved or the testing can be done incompletely and the unexecuted tests can be executed after the delivery.

4.4.2 Risk analysis

Another method to decide which test cases to run is to use risk assessment. Risk is formed by to attributes: the probability of the occurrence and the severity of impact. When using this method, the test cases are not evaluated as such but the potential problems tested by those test cases are, so the first order of business is to list the potential problems. Like with the priority range, any range of values that seems suitable can be given to both the probability and the impact of the problem, as long as the criteria are the same for the period of risk assessment. If there is no clear or formal way to determine these attributes, a good way to start is to find the problems with the highest and lowest values first and then assign the rest of the problems some values according to the guidelines set by the established extremes.

These two attributes can be used to categorize the problems and use this knowledge to select the test cases that will be executed. One way to use them is to calculate the risk exposure by multiplying the values together. The bigger the risk exposure, the more important it is to test the potential problem.

Louise Tamres presents a risk matrix /4/ as another example on how to apply the probability and impact attributes to determine the important problems that have to be tested. The idea is to place the problems into a risk matrix and use their position in it to make that determination. She states that typically a risk matrix contains four quadrants, each representing a priority class. These priority classes are defined as:

1. High severity and high probability.
2. High severity and low probability.
3. Low severity and high probability.
4. Low severity and low probability.

The risk matrix (Figure 8) can be used to weight either probability or severity easily by switching the locations of priority 2 and priority 3.

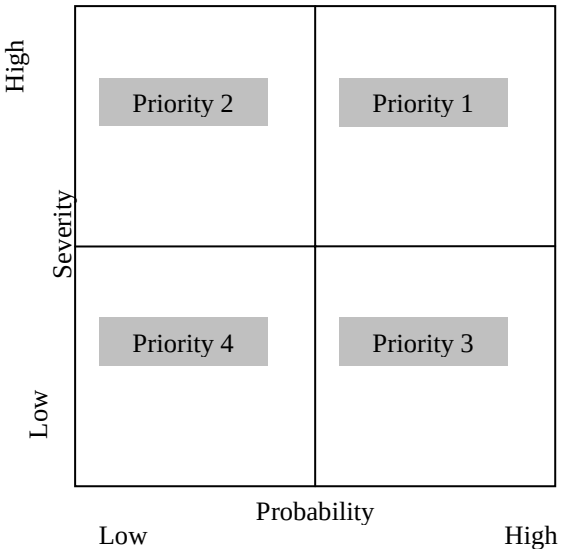

Figure 8: Risk matrix.

Louise Tamres also presents some other ways to divide the matrix, depending on what the organization feels should be weighed in assessing the importance of test cases. If the probability and the impact of the problem are of equal weight, then the matrix might be divided into diagonal bands, shown on the left side of Figure 9. Or if the weight should be placed upon all the critical problems without any regard to the probability, the matrix can be divided into five areas, depicted on the right side of Figure 9. /4/

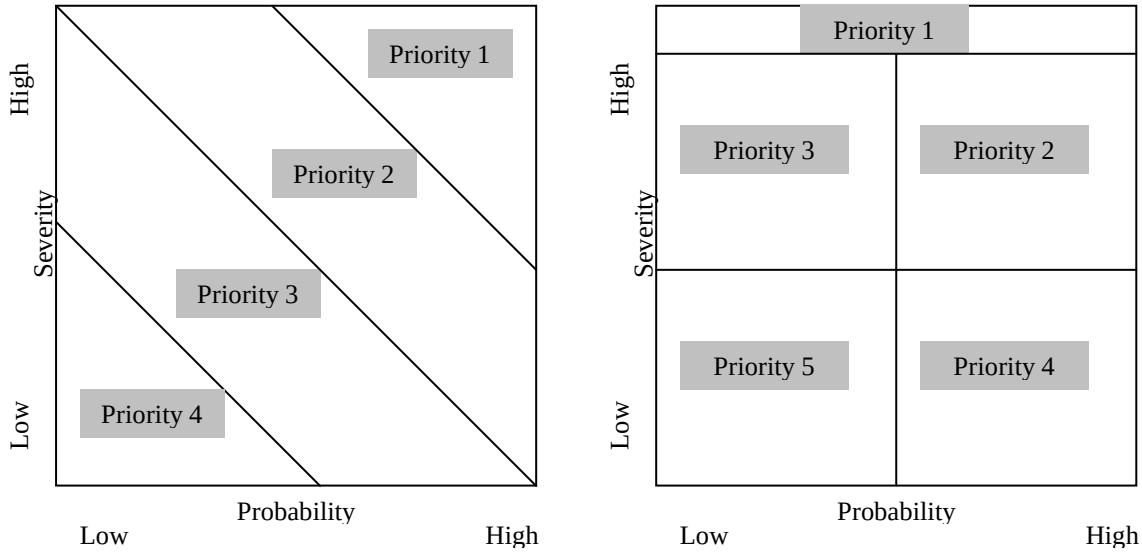

Figure 9: Possible ways to divide the matrix into different thresholds.

5 Software testing in practice

I started my career at Necsom Ltd. as a software engineer, but gradually I started to receive system administration tasks related to our test laboratory and operational network on the side. My principal responsibility however was software development until the spring of 2003 when I was given the opportunity to move to software testing and to write my master's thesis. The project that is the basis for this thesis was to study software testing in general and come up with a proposal on how the software testing process could be improved at Necsom Ltd.

5.1 Tools and supporting software

Necsom utilizes a number of different 3rd party applications and systems in software development and testing. The most useful and important of these are described shortly in this chapter.

5.1.1 Concurrent Versions System

Because there may be, and almost always are, many individual programmers developing the same product a version controlling system is essential for all such software development projects. It stores all the files in the project and maintains history data on the changes made to these files. In version controlling systems, all the files are kept in a central repository, to which the files are checked in or checked out from. CVS, or Concurrent Versions System, is a widely used version controlling system, also used in all the Necsom's software projects.

When using CVS, checking out a file does not give a developer exclusive rights to that file. Other developers can also check it out, make their own modifications, and check it back in because CVS detects when multiple developers make changes to the same file and automatically merges those changes. However CVS is cautious and will merge

automatically only as long as the changes are not made to the same lines of code. If CVS cannot safely resolve the changes, the developer will have to merge them manually. /17/

Why is version control important to testing? The answer is very simple: you absolutely have to know what you are testing. If you cannot be certain that you are testing the newest version of the code in the repository, chances are you executed the tests with some obsolete version of the software that never worked properly and end up reporting nonexistent bugs. Or even worse: the older version may have worked once, but new changes in the code have rendered the tested functionality inoperable. In such a case all your testing would show that everything is working correctly, even if the application could no longer even be started properly.

5.1.2 Build-tool for java

I recommended that one of the software projects start to make use of Ant instead of Make for compiling and building its packages, and so far it has proven to be a versatile and easy to use build-tool for java, fulfilling the promises given by Erik Hatcher and Steve Loughran:

In order to build a software product, we manipulate our source code in various ways: we compile, generate documentation, unit test, package, deploy, and even dynamically generate more source code that feeds back into the previous steps. These steps are initially done manually, but when we tire from doing the repetitive, we look for existing tools that can ease the burden of repetition. Ant is a Java-based build tool, designed to be cross-platform, easy to use, extensible and scalable. /18/

Although JUnit, the de facto unit testing API for java development, is not yet taken into use with current projects at Necsom Ltd., Ant is integrated with it and can easily be used with JUnit to automatically execute test suites during the build process.

5.1.3 Build-tool for C and C++

Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files. /19/

It is another widely used build-tool that is especially suitable for projects using C and C++, but it also works well with other programming languages, like Java which is used in all our software projects. Make is still used on older software projects at Necsom Ltd., but it will likely be replaced by Ant in the near future because Ant is more suitable for use with Java than Make.

5.1.4 Java scripting tool

BeanShell is a small, free, embeddable, Java source interpreter with object scripting language features, written in Java. BeanShell executes standard Java statements and expressions, in addition to obvious scripting commands and syntax. BeanShell supports scripted objects as simple method closures like those in Perl and JavaScript(tm). /20/

Java BeanShell is used for unit testing some of the modules in our software projects. For this to be possible the programmer who implemented the module under testing has to implement a Java BeanShell test driver. This test driver creates an instance of the module to be tested and can then be used to call all the methods available in the module. If these unit test programs are kept up to date, Java Beanshell can easily be used to automate unit testing with the help of Java Beanshell scripts. At the moment the results still have to be analyzed manually, but an automated result analyzer is being developed.

5.1.5 Defect tracking system

When the project to improve the testing process began, Necsom Ltd. had deployed a different defect tracking system, but because we had problems and were generally

dissatisfied with it we decided to migrate to Bugzilla, which has proven to be one of the most important testing support tools currently in use at Necsom.

Bugzilla is one example of a class of programs called "Defect Tracking Systems", or, more commonly, "Bug-Tracking Systems". Defect Tracking Systems allow individual or groups of developers to keep track of outstanding bugs in their product effectively. /21/

Bugzilla keeps all the information about found defects in a single database, from which this information can be queried or modified using a web browser. The installation and setup of Bugzilla was easy and pretty straightforward and the biggest problem was the migration of existing defects from the old system to the new one which was done by manually entering all the defects into Bugzilla.

5.2 Organization and roles

There are 11 employees working in the research and development section of Necsom Ltd. (which does not include the upper management). Before the effort to design and reorganize testing the staff was organized roughly as presented in Figure 10.

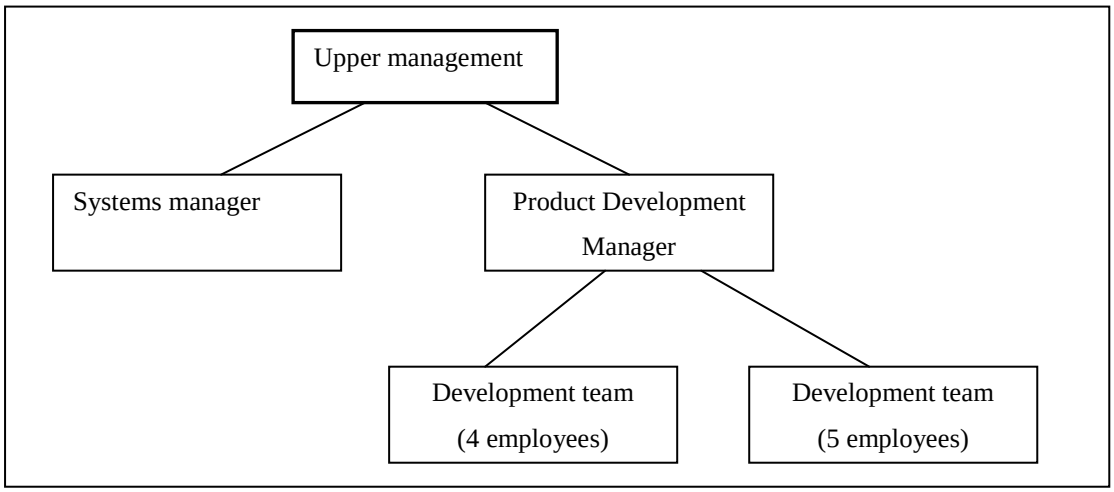

Figure 10: Organization before any emphasis was given to software testing.

As can be seen from Figure 10, the testing of the software was not any consideration when the staff was originally organized, which fits the description for the case number

one in Figure 4 where the testing is each unit's responsibility. This resulted almost constantly in scheduling problems, even if the testing was included in project plans and master schedules, because the same people who programmed the different modules also had to test the software and fix all the bugs found and as is often the case, testing took the back seat when there were bugs to be fixed. This is one of the problems predicted for this organizational approach.

At times reporting was also chaotic, because it was easier to fix all the simpler bugs straight away and not make any bug entries for it. This made it very difficult to know what changes were made between different releases. There were some occasions when a bug for an older version was reported and someone had already fixed this bug in the newer versions of the software without documenting that fix anywhere. Because the person responsible for fixing this bug could not know when this bug fix had been introduced he had to first make sure that the bug really did exist (this included the installation and configuration of software and the environment for it), discover the root cause and then proceed with fixing it. All this would have been much easier if the developer had been able to check if that same bug had already been found and fixed for later versions and either had the customer update the software or just copy the already existing fix and issue a patch. Not to mention that without the documentation informing the bug was fixed, the programmer also had to check all the newer versions of the code to make sure that if this same bug existed for them too, it was also fixed for those newer versions of our software.

To address these issues and to enhance the testing of software produced at Necsom Ltd. attention was given to testing and the organization was restructured. Having only 11 employees, who already had their work cut out for them placed quite severe limitations on what could be done. Therefore the reorganization resulted in something of a mix between cases 2 and 4 pictured in Figure 4. I moved completely to software testing and was temporarily given the tasks of a testing manager for the purpose of designing software testing since the development manager was already overburdened by his responsibilities at the time. Because the company could not afford to reassign any more people from development to testing some of the programmers are still required to wear

both the hats of the developer and the hat of the tester. This is not the best imaginable situation, but we had to be satisfied with it.

A two stage introduction of testing to the existing organization was planned. The first stage is implemented according to the scheme depicted in Figure 11. The second stage of the reorganization will be implemented at a later date when there are more human resources available to fill the positions required by it. The second stage, which has not yet been implemented, is presented in Figure 12.

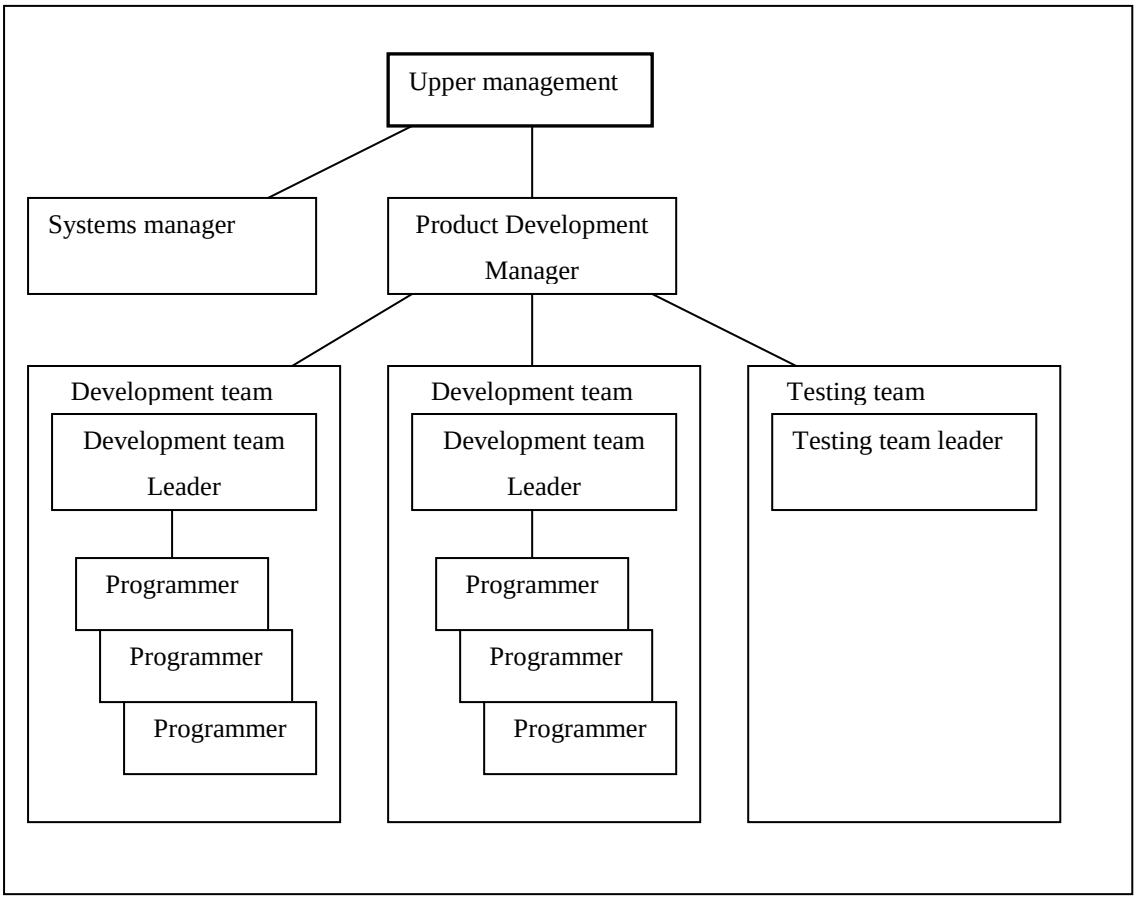

Figure 11: The first stage of reorganization.

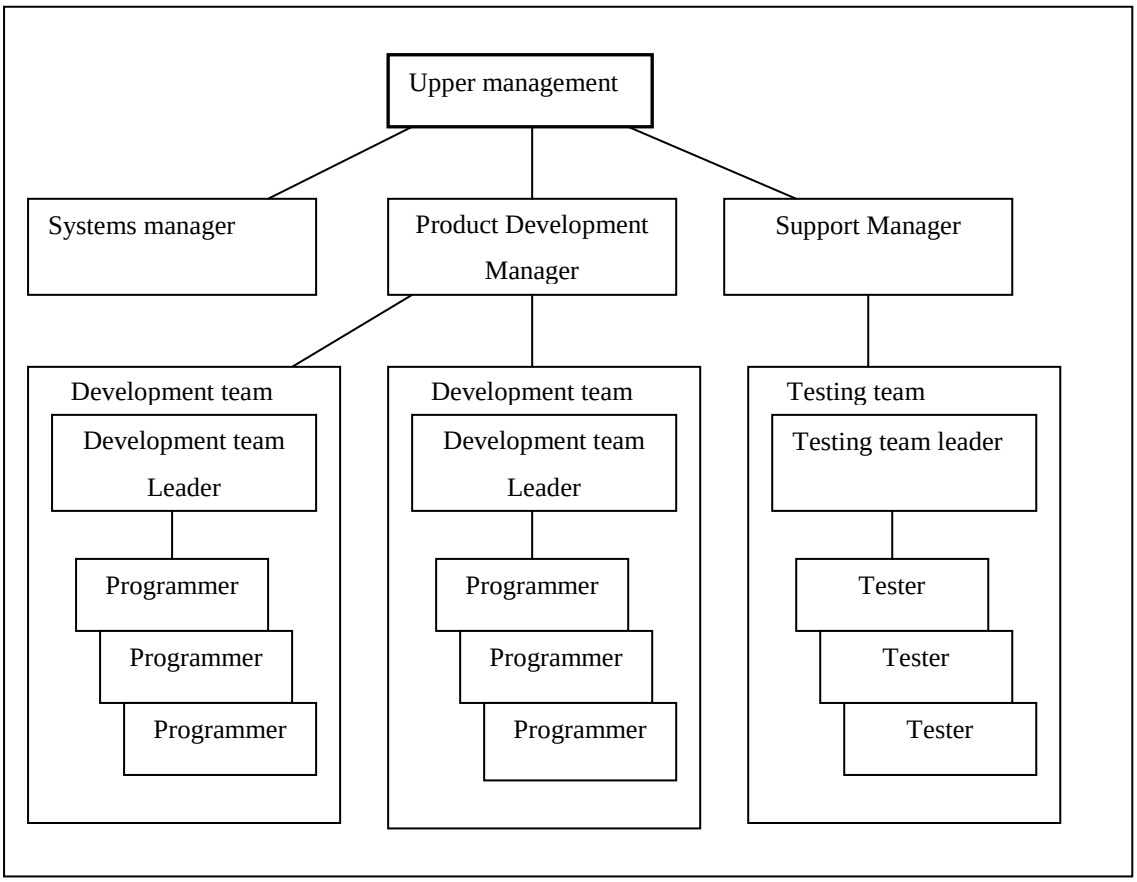

Figure 12: The planned second stage of reorganization.

5.2.1 Testing manager

At the first stage the product development manager also handles the testing manager's responsibilities when my temporary assignment as testing manager ends. This means that he manages and supports the testing team, arranges necessary training and does his best to acquire necessary resources for testing, including human resources, hardware and software. In order to acquire these resources he liaises with the upper management and informs them once a week about testing based upon the reports on the progress of testing from the test team leader and the independent observer.

The testing manager (either the product manager or the support manager at Necsom) arranges for an independent observer to be present for at least some of the tests executed and receives his reports concerning that monitoring. The reports from the team leaders

and observers also contribute to the evaluation of the testing process by pinpointing possible problem areas that can be improved for future projects.

The testing manager is also the point of contact for the customers using our software who require support or who have problems they want to report to Necsom Ltd. The testing manager first checks these problems and then assigns them to the appropriate people who will be responsible for resolving these issues. The testing manager is responsible for overseeing that these issues are handled properly.

During the first stage, all of the above tasks are in addition to the tasks of the development manager which would be a fulltime job in itself. Since the above task list assumes that the test team manager does not have any other duties but managing the test teams, the testing manager has to delegate some tasks ordinarily appointed to him to the test team leader just to cope with things.

5.2.2 Test team leader

The test team leader is responsible for the day-to-day running of the test project. He allocates the tasks to the testers under his supervision. It is his task to design the test plans for each testing phase and to submit these plans to the testing manager for reviewing.

The team leader maintains contact with both the testing manager and the development team leader so that they are kept up to date with the current situation and know of the problems encountered and they can inform the test team leader of any new changes or error fixes introduced to the software under testing.

The testers report to the test team leader at least twice a week or immediately if they encounter problems that have to be solved before the testing can be commenced again. The team leader collects these reports and generates a report on the progress of testing and delivers his report to the testing manager. This report must contain the test cases

executed and any problems encountered during the testing, including problems found during test case execution as well as problems with the test environment or testing staff.

Because of the lack of available personnel during the first stage, the team leader is required to help the testing manager with the testing process planning, evaluation and improvement, task identification and effort estimation. Also for the same reason the test team leader has to perform tasks that would otherwise fall to the domain of test analysts and testers.

5.2.3 Test analyst and tester

During the implemented first stage of reorganization test analysts and testers are drawn from the ranks of the development team, preferably so that a few of the developers only execute the test cases, while the rest are focused on fixing the problems found and if situation permits, designing the next version of the product.

These testers and the test team leader will plan and execute test cases for integration, system and acceptance testing and generate reports on the progress of testing. Every time a test case is executed, a test diary for the ongoing testing stage is updated. In addition to the test diary, the testers also report to the test team leader twice a week.

5.2.4 Observer

When looking at the Figure 11 it is obvious that at Necsom Ltd there is only one person who is both available and independent from software development and testing: the independent observer defaults to the systems manager. During the first stage, the systems manager should keep contact with the product development manager and the test team leader to be briefed on the stage of testing and the times when he should be present. Since there is only one observer, it cannot be assumed that he is always available. We try to offset this by having the observer monitor some tests randomly and unannounced when he finds the time for it.

For every test monitored, the observer generates an observation report, stating his views on how the testing was done: test executed, outcome of the test, were the instructions followed correctly, were the results recorded properly and any irregularities encountered.

5.3 Testing

Necsom Ltd. uses four levels to divide the testing process into more easily manageable elements. These levels are as follows:

1 Unit testing.
2 Integration testing.
3 System testing.
4 Acceptance testing.

A specific test plan is produced for integration testing, system testing and acceptance testing phases. In general, the test plans must contain at least the following things:

- Human resources available during the testing phase.
- Objectives of the testing at the stage the plan is made for.
- Deliverables.
- Strategy used in the testing phase:
    - the main focus of testing.
    - how the testing is divided into smaller steps, detailed in test suites.
- Testing exceptional and error situations.
- Tasks required for the testing phase in question:
    - Planning the test cases.
    - Test environment preparation.
    - Reviews for the test plan and the individual test cases.
    - Executing the test cases.
    - Writing the test report.
- Scheduling the tasks for the testing phase.
- Entry and exit criteria.
- Risk assessment and control.

5.3.1 Unit testing

Unit testing is performed by the programmers who originally coded the unit which is under testing. Although a separate test plan for unit testing often is not required, I felt that some sort of plan is required for unit testing to make it possible for other people to test the same units even after the coder is no longer available to do the testing for his own component. Therefore the first task of the developer when he starts unit testing is to generate a simple unit test plan for his unit, which contains at least the following items of interest:

- Description of the test environment.
- Short description of the unit (may be as simple as the methods provided).
- Test cases that will be executed.
- Appendix, containing all the other information that is necessary for other people to execute the same test cases if necessary e.g., commands used with the test driver.

These unit test plans are not to be confused with test plans for other testing phases, such as integration test plan, since they are created for individual units and lack many things that are included in all the other test plans. The unit test plans are not reviewed, but they should be so accurate that somebody else in the development team can execute the unit tests according to the plan.

It is recommended that these tests are designed and implemented according to the example offered by an already implemented simple test driver. This test driver uses Java Beanshell so that it can call individual methods that are implemented in the unit under testing. This very often requires also some stubs to be implemented for all the adjoining components that are not yet available.

Unfortunately the lack of resources often forces us to do the unit testing the old fashioned way we have used earlier: editing the code of the unit for testing purposes. This is not an ideal situation, since the code that is being tested is not exactly the same as the final code that will be delivered with the final product. This way some of the error situations may

even be checked more easily than in any other way, but there is always a risk involved with this method. It is possible that an error is made when making such a change and everything seems to work fine until the unit is tested in the real environment with other fully implemented components that are part of the same application.

When analyzing the results of the executed unit test cases the testers rely heavily on the log files the units generate and any error message that are printed to the console from which the code is being executed. When errors in the code are noted, they are immediately inserted into Bugzilla, our chosen bug tracking system. When a fix for that particular error is implemented and tested, the status of the error report in the error database is changed accordingly.

Test reports for the unit tests are not generated to keep the amount of effort required to a minimum and therefore the only documents that remain after unit testing are the unit test plan and the error reports in the error database.

5.3.2 Integration testing

Integration testing is done by the test team leader and those developers that are available for testing during the time scheduled for integration testing. The first requirement for entering the integration test phase is that the test team leader has written the integration test plan and it has been reviewed and approved by his superior (either the product development manager or the support manager). Depending on the project and the exact organization related with it, customers may have a representative who is responsible for quality assurance on their side take part in the reviewing of the test plan.

When this is done the testers may start designing all the different test suites. At Necsom Ltd. these test suites are simply sets of individual test cases that have some common nominator between them e.g., internal communication between different units or some other common functionality. The test team leader reviews and approves these test suites and this approval is the second requirement for starting the integration testing.

The third requirement for starting the execution of integration testing is that most of the implementation is ready, unit tests have been executed and all errors found that would prevent integration testing are fixed. It would be preferable that all the implementation tasks are successfully completed before starting integration testing, but again life has shown that often it just is not practical to wait for all the implementation tasks to be 100 percent ready before starting testing. If all the implementation tasks are not yet done when the testing should be started, the product development manager, development team leader, support manager and test team leader must have a meeting where they decide when the integration testing can be started.

When these three requirements have been met, the testers can start executing the test cases. The executor of the test case does not have to be the same person who designed it in the first place but this is often the case at our company. All the errors encountered during the testing are entered into the bug database and the integration test diary. This test diary is implemented as a simple datasheet that contains fields for the test case identifier, date the test was executed at, whether the test was successful or not and a description field for any important notes about the test case (like bugs found, errors in the test case itself, problems with test execution and so on). All critical bugs should be in either resolved, verified or closed state before the integration testing phase can be ended.

The deliverables for integration testing are the reviewed integration test plan, reviewed test suites, integration test diary, tested code and an integration test report prepared by the test team leader at the end of integration testing. This integration test report must have a summary of defects found during the testing, the current state of those defects and the status of all the exit criteria defined for the integration testing phase.

5.3.3 System testing

The system testing is done by the same people as the integration testing: the test team leader and the testers drawn from the development team. Before the system testing can be

started the system test plan must have been written by the test team leader, reviewed, and approved by the testing manager. If the test plan is ready and the requirements for the software are available, the test team can start to prepare the test suites.

When the test suites are approved and the whole system is completely implemented the system testing can commence. However once again it has been my experience that the latter of these requirements has to be occasionally set aside and some of the system tests may be executed when all the necessary components required to run the test cases are complete. If this is the case, some of the test cases have to be re-executed to validate that all the features work as specified when all of the components are finally integrated together.

A system test diary, similar to the integration test diary, is updated throughout the system testing phase. This diary, along with the approved system test plan, system test suites, reviewed and tested user documentation, the system test report written by the test team leader and the system tested software are the deliverables for the system testing phase.

5.3.4 Acceptance testing

Acceptance testing is basically a subset of test cases selected from all the test cases prepared for system testing. Both Necsom and the customer give input as to which test cases are adequate for the acceptance testing. If the customer feels there is a need to add some new test cases for the acceptance testing before their approval, this can be negotiated at this point. It would be preferable to get at least some advance notification of this as early as possible in the test planning stage, so that these tests could be already included in the system testing, but practice has shown that customers often come up with some specific tests they want to execute pretty late in the software development cycle. Based on the requests of the customers and his own consideration the test team leader prepares an acceptance test plan, which is then reviewed and approved by the product development manager (or support manager). The test team leader also compiles the test

suites necessary for acceptance testing based on the requests of the customers and test suites used in system testing.

Prerequisites for acceptance testing are the accepted test plan and fully implemented and system tested software. In addition all the critical errors discovered should be corrected. Acceptance testing can be done either in our test laboratory, or in the customer's premises with the environment that they provide, depending on the preference of the customer.

The deliverables for this phase are the approved acceptance test plan, acceptance test suites and acceptance tested product.

5.3.5 Regression testing

Regression testing is currently done on a case by case basis. The product manager and test team leader decide which test cases have to be executed after the code of the software has changed. Test cases are selected from unit testing, integration testing and system testing phases. It is the product manager's responsibility, with the help of the test team leader to include regression testing in the overall schedule for the project.

5.4 Test cases

A common template for the test cases has been created and made publicly available inside Necsom Ltd. (see Appendix 1). This template is only an example and should always be modified to meet the needs of the particular project it is used in. I added the following fields to the basic requirements for test cases:

- Execution date
- Outcome of the test (success/failure)
- Test suite this test case belongs into
- Priority (on a scale from 1 – 3, 1 being the most important)

- Components touched by the test case
- Descriptive name for the test case
- Space for comments about the test in general (use is not mandatory)

5.5 Test environment

The test environment is one of the things redesigned (Figure 13) during the project to improve testing at Necsom Ltd. The old configuration of the test laboratory was found to be too restricting and it did not support all the tests that were deemed necessary, the most important issue being the lack of a firewall in the test environment. The requirements for the new testing environment were that it had to include multiple routers and subnetworks, a firewall, and numerous hosts running different operating systems (Linux, HPUX, Solaris, Windows NT and Windows 2000) and the following IP-based services: HTTP, HTTPS, DNS, FTP, IMAP, LDAP, NTP, POP, Radius, SMTP, Telnet, WAP and PAP.

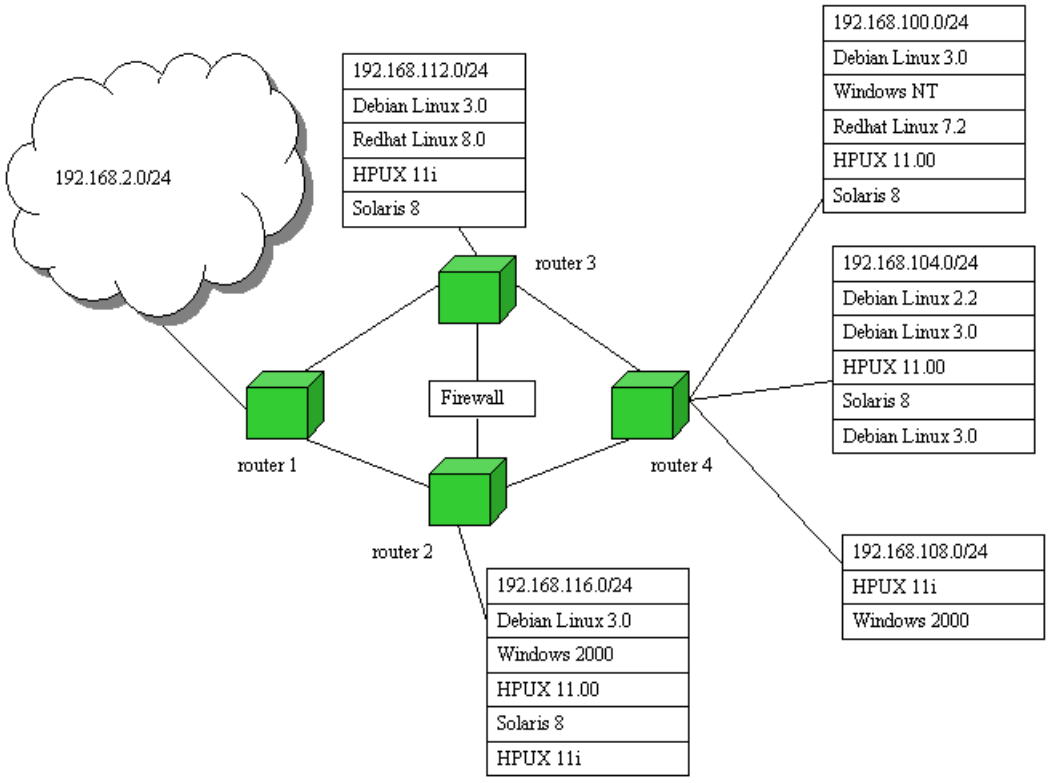

Figure 13: Topology of the Necsom test laboratory.

The 192.168.2.0 network is the production network used by the development teams and it contains all the necessary services for a software company to carry on its day-to-day operations, including email, web, CVS and bugzilla servers.

Networks 192.168.112.0 and 192.168.116.0 had the more powerful computers available and the hosts in them contained services necessary for the operation of the test network, especially the DNS and DHCP servers. The hosts in these networks are also used to run the applications developed at Necsom for testing purposes. Also of note is that these two networks are separated by a firewall, which actually is a bridge between the networks and which also works as a firewall. This way neither of the networks is able to see the firewall host, but it still can block the traffic between these separate networks. If the traffic from these networks is destined to any of the remaining three networks (192.168.100.0, 192.168.100.4 or 192.168.108.0) and the traffic has to pass through a firewall, this can be accomplished by static routing.

Networks 192.168.100.0 and 192.168.104.0 are used to host all the required IP-based services that are not necessary for the operation of the test network. The computers in these networks are generally low-end computers ranging from Pentium 166 MHz to Pentium II 266 MHz with an occasional Ultra10 and one HP-9000/859.

The last of the networks is 192.168.108.0 and it was added when it became obvious that the testing in some of the projects require very specialized hardware and software which has to be supplied by the customer for testing purposes.

5.6 Problem reporting

The problem reporting system of choice at Necsom is Bugzilla. The Bugzilla has a web interface, compatible with all the common web browsers, for entering new problem reports into the problem database. This is a strongly recommended method of generating new problem reports in order to generate problem reports formatted in a unified way that contain all the relevant information. It was noticed during our earlier projects that the

problem reports generated by our customers were often quite haphazardly constructed and lacked a lot of the information necessary to reproduce and prioritize the problem. The web interface made it easy to open access for our customers to the Bugzilla and they learned quite quickly the most important aspects of using it. Still we sometimes get problem reports through other channels, for example by email, that do not conform to the problem reports created using Bugzilla interface, the recipient of the mail has to check the report, possibly ask for more information and then enter all this into the problem database.

Our problem reporting interface is a HTML form with all the necessary fields required for an accurate problem report. Most of these fields have a set of values that can be chosen from a drop down list when generating the problem report. This feature makes it possible for laymen to use our problem reporting system to enter their own problem reports. Naturally these reports have to be analyzed by our staff to ascertain that all the information is correctly entered and in line with our own assessments. For example the severity of some error might be viewed as normal by our customer, whereas we would set a lower severity to for it. The analyzer makes the necessary corrections, and sets the severity to a value that is in line with the other problems reported. Table 1 shows the seven possible severities in a descending order from the most severe to the least severe.

Table 1: Problem report severities.

Blocker
Critical
Major
Normal
Minor
Trivial
Enhancement

The form used to generate the problem reports has the following fields: Product, version of the product, component with the problem, Hardware environment, Operating system, and Severity. All these fields have menus from which the selection is made. In addition to

these there is a name field (for a short descriptive name of the problem) and a description field (text area) where the detailed description of this problem is entered. The forms have also preset groups that can be allowed or disallowed to have access to the problem report generated. When the problem is submitted into the reporting system, it automatically generates emails to predetermined people (currently product development manager, development team leader and test team leader) informing them about the problem.

In addition to the fields mentioned above the problem reports have a priority field (on a scale from 1 to 5) and six different states: new, assigned, reopened, resolved, verified and closed. The resolved, verified and closed states alone are not enough and they have additional information that is contained in the resolution field which gives a reason why the status of a problem is closed. Possible resolutions for a problem are: fixed, invalid, wontfix, later, remind, duplicate and worksforme.

The life cycle of a bug report proceeds normally from the moment it is introduced to the defect tracking system with its initial state “new”. When it is analyzed it will be assigned to a person who will be responsible for dealing with it and the state will be changed to “assigned”. When the issue is addressed a resolution is set for the problem and the state changes to “resolved”. All the resolved issues have to be checked by a tester and if the issue is dealt with acceptably the state will be changed to “verified”, otherwise the state will remain as “assigned”. The product development manager and the testing manager have the authority to change the state of a verified problem report to “closed”. Because it is possible for the same issue to resurface, the “closed” state may be changed to “reopened” and it is assigned back to the person who was responsible for it in the first place. From reopened the lifecycle of the problem report will continue the exact same way it would if the state was assigned. Figure 14 shows the life cycle of a problem report.

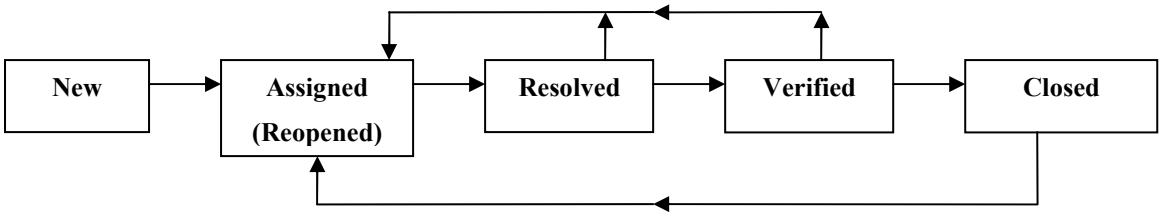

Figure 14: The life cycle of a problem report.

Files can be attached to generated problem reports and inserted into the defect database. These attachments are usually log files, screenshots or test cases that produced the error, but for example also links to web pages can be attached.

Bugzilla automatically monitors the state of the problem reports, and there are time limits defined for how long a problem can be in certain states before the system generates an email reminding the people involved with the report about it. For example no problem report should be in new state for longer than 3 days.

6 Conclusions

During this project it quickly became very clear that the claim about testing being a complex process requiring a detailed and organized approach that should not be limited only to executing the tests on the software hit the mark very well. The testing should never be limited to only executing the tests: the proper planning of testing is essential to improve the quality of the testing process and for building good and efficient test cases. Also it is much better to have appropriate verification activities like formal reviews that catch the errors very early in the process, preferably during the initial planning in the development life cycle. Neither should the testing be limited to the software itself, but it should also include the user documentation, specification documentation, design documents, test plans, test cases and the testing process itself in its scope. The testing has been planned at Necsom Ltd. before, but the quality of our software development process was improved when more formal reviews were conducted upon the plans and test cases generated.

When I was given the mandate to improve testing done at Necsom Ltd., the company was already using a slightly modified waterfall model as a basis for the development process. Because of this the switch to doing the testing according to the V-model was quite readily done. Biggest problems encountered were the organizational changes and new tasks some of the people had to become accustomed with. The test environment provided by Necsom Ltd. is now flexible enough and it meets our needs very well. If no new projects are started and no new major services have to be setup for testing purposes, the test laboratory does not need any other work than maintenance in the near future. It also proved to be a good choice to deploy the new problem reporting system, since now the problem reports are better than in the past projects and their state is more easily tracked throughout the process.

Having now observed how our testing has proceeded according to the outlines set in this thesis it has become obvious that our testing today is more effective than before, but it still should be improved further. Currently there is too much work to do for the test team

leader and in the future Necsom should continue with the reorganization either by starting the stage 2 described for organization reform or by organizing the staff in some other manner. The development process evaluation and regression testing are two other things that should be systematically planned and implemented by Necsom Ltd. in the future.

References

/1/ Simons, Geoff. Introducing software engineering. The National Computing Centre Limited, 1987. ISBN 0-85012-593-6.

/2/ Software QA and Testing Resource Center – FAQ Part 1. http://www.softwareqatest.com/qatfaq1.html#FAQ1_2 1.8.2003

/3/ Prof. Dr. Stefan Leue. IT 460 Software Engineering. http://tele.informatik.uni-freiburg.de/~leue/IU/it460.part1.pdf 1.8.2003

/4/ Tamres, Louise. Introducing software testing. Pearson Education Limited, 2002. ISBN 0-201-71974-6.

/5/ Kit, Edward. Software testing in the real world: improving the process. ACM Press, 1995. ISBN 0-201-87756-2.

/6/ Kaner, Cem. Lessons learned in software testing: a context driven approach. John Wiley & Sons, Inc., 2002. ISBN 0-471-081122-4.

/7/ GlobalTester (Verification and Validation) http://www.globaltester.com/sp1/define_vv.html 30.7.2003

/8/ Integration Testing by QaAssociates Software Quality Solutions. http://www.qaassociates.com/qacorner_june02.htm 1.8.2003

/9/ http://www.satisfice.com/presentations/strategy.pdf 5.8.2003

/10/ Watkins, John. Testing IT an off-the-self Software Testing Process. Cambridge University Press, 2001. ISBN 0-521-79546-X.

/11/ Beizer, Boris. Software testing techniques. 2nd edition. International Thomson Computer Press, 1990. ISBN 1-850-32880-3.

/12/ System testing. http://www.comp.lancs.ac.uk/computing/resources/ser/syseng/cbse/testing.ppt 1.6.2003

/13/ comp.software.testing Frequently Asked Questions (FAQ) http://omicron.felk.cvut.cz/FAQ/articles/a1083.html 1.6. 2003

/14/ The Various Types of Software Testing Used in Today's IT World. http://www.mcdonaldbradley.com/HTML/white_papers/Types_0f_Testing.pdf 2.6. 2003

/15/ Usability, Usability Testing Frequently Asked Questions. http://www.usabilitysciences.com/services/faq1.html#Usability_Basics 1.6.2003

/16/ An introduction to software testing. http://www.iplbath.com/pdf/p0820.pdf 2003. 5.5.

/17/ CVS for new users http://www.cvshome.org/new_users.html 8.6.2003

/18/ Erik Hatcher, Steve Loughran. Java Development with Ant. Manning Publications Co., 2003. ISBN 1930110588

/19/ GNU Make – GNU Project – Free Software Foundation (FSF). http://www.gnu.org/software/make/make.html 21.6.2003.

/20/ Beanshell – Introduction -. http://www.beanshell.org/intro.html 21.6.2003

/21/ Bugzilla Project – What is Bugzilla? -. http://www.bugzilla.org/about.html 21.6.2003

Appendix A: Test case template

Product name	**Execution date:**
Testing phase	**Outcome:**

Test suite:

Test ID:

Test Name:

Test Priority:

Test Environment

- Operating system
- Hardware and software requirements

Components

- All the relevant components to the test

Prerequisites

- Necessary actions before the test case can be executed

Test Execution

1. detailed instructions for the test case execution

Expected Results

- what is the outcome of a successful test

Test Results

Comments