# AUTOMATED BASIC TESTER

LAHTI UNIVERSITY OF APPLIED SCIENCES
Degree Programme in Information Technology
Software Engineering
Bachelor's Thesis
Spring 2009
Janne Kankola

Lahti University of Applied Sciences
Degree Programme in Information Technology

KANKOLA, JANNE: Automated Basic Tester

Bachelor's Thesis in Software Engineering, 57 pages, 8 appendices

Spring 2009

ABSTRACT

---

This thesis was made for Oy L M Ericsson Ab as a part of larger project to automate the testing of Ericsson Network IQ (ENIQ). The purpose of this thesis was to develop a test automation tool for the basic testing of ENIQ Technology Package. The scope of this study was limited to automating loading tests, as a separate study was made about automating the verification report testing. The main goals were to get the test tool into use in the specified time frame and to be able to test all Tech Packs that use MDC data type.

ENIQ itself is programmed using Java and it uses Sun Solaris 10 as an operating system. The testing tool was designed to be run from a Windows environment as every developer has a Windows workstation and because Windows was more familiar to the developers than Solaris. It was decided that some of the functionality should be implemented using external tools. The testing tool was also programmed using Java as Java is widely used within Ericsson, which makes the future development of the software easier.

Manually testing the loadings is really difficult and time consuming if done thoroughly. It is fairly easy to test that something has been loaded into the database but it has been practically impossible to test loadings with 100% coverage and accuracy. With the help of the test automation tool full coverage and accuracy can be achieved easily.

For the automation to work, the testing tool needed to connect from the Windows workstation to the server running Solaris using SSH protocol. When the testing tool was connected to the server, it had to execute certain commands and transfer the input files from the server to the Windows workstation, before the actual testing could begin.

The results of the automation were really positive; the testing became faster even though testing was more thorough and the testing tool was able to catch minor bugs that had existed for a while. With the help of the tool the test can now be run overnight or during weekends, which increases productivity.

Keywords: testing, automation, Ericsson Network IQ, Java, Solaris

Lahden ammattikorkeakoulu
Tietotekniikan koulutusohjelma

KANKOLA, JANNE: Automated Basic Tester

Ohjelmistotekniikan opinnäytetyö, 57 sivua, 8 liitesivua

Kevät 2009

TIIVISTELMÄ

---

Tämä opinnäytetyö tehtiin Oy L M Ericsson Ab:n tarpeisiin osana suurempaa projektia, jonka tarkoituksena oli automatisoida Ericsson Network IQ:n (ENIQ) testausta. Työn tarkoituksena oli kehittää automatisoitu testaustyökalu ENIQ:n Technology Package:ien basic-testaukseen. Automatisointi rajattiin käsittämään ainoastaan tietojen latauksien testaaminen, sillä verifiointiraporttien testaamisen automatisoinnista tehtiin erillinen opinnäytetyö. Työn päätavoitteena oli saada testaustyökalu käyttöön määritellyssä ajassa ja pystyä testaamaan Tech Packit, jotka käyttävät MDC tiedostotyyppiä.

ENIQ on ohjelmoitu käyttäen Java-ohjelmointikieltä, ja sen käyttöjärjestelmänä toimii Sun Solaris 10. Testaustyökalu kehitettiin ajettavaksi Windows-ympäristöstä, koska kaikilla kehittäjillä on käytössään Windows työasema ja koska Windows oli kehittäjille tutumpi ympäristö kuin Solaris. Osa testityökalun toiminnallisuudesta päätettiin toteuttaa käyttämällä ulkoisia ohjelmia. Myös testityökalu ohjelmoitiin Java-ohjelmointikielellä, koska Java on yleisesti käytössä Ericssonilla, mikä helpottaa testityökalun jatkokehitystä.

Latausten testaaminen manuaalisti on erittäin vaikea ja aikaa vievä prosessi, jos se tehdään perusteellisesti. On suhteellisen helppoa testata, että jotain on onnistuttu lataamaan tietokantaan, mutta on ollut käytännössä mahdotonta testata lataukset 100% kattavuudella ja virheettömyydellä. Testityökalulla täysi kattuvuus ja virheettömyys saavutettiin helposti.

Jotta automatisointi olisi mahdollista, testaustyökalun täytyi muodostaa yhteys Windows-työasemalta Solaris-palvelimelle käyttäen SSH-protokollaa. Kun yhteys oli muodostettu, testaustyökalun täytyi suorittaa palvelimella tietyt komennot ja siirtää syöte tiedostot palvelimelta Windows-työasemalle, ennen kuin itse testaaminen voitiin aloittaa.

Testauksen automatisoinnilla saavutetut tulokset olivat erittäin positiivisia. Testaus nopeutui, vaikka testit olivat paljon syvällisempiä ja testityökalulla löytyi myös muutama vähäpätöinen bugi, joita ei oltu huomattu aikaisemmin muissa testeissä. Testityökalun ansiosta testit voidaan ajaa öisin tai viikonloppuisin, mikä puolestaan lisää tuottavuutta.

Avainsanat: testaus, automatisointi, Ericsson Network IQ, Java, Solaris

CONTENTS

ABBREVIATIONS

| | |
|---|---|
| 3GPP | 3rd Generation Partnership Project. A collaboration between groups of telecommunications associations. |
| ABT | Automated Basic Tester. A test automation tool developed as part of this thesis. |
| ASCII | American Standard Code for Information Exchange. A coding standard that can be used for interchanging information, if the information is expressed mainly by the written form of English words. |
| ASN.1 | Abstract Syntax Notation One. A flexible notation that describes data structures for representing, encoding, transmitting and decoding data. |
| AT&T | AT&T Inc. The largest provider of both local and long distance telephone services in the United States, which also provides digital subscriber line Internet access and wireless telephone service. |
| ATM | Automated Teller Machine. A computerized telecommunications device that provides the customers of a financial institution with access to financial transactions in a public space without the need for a human clerk or bank teller. |
| AdminUI | Administrator User Interface. The main control interface of Ericsson Network IQ. |
| BIT | Basic integration testing. A testing phase where all the changed software modules are integrated into the same environment. |
| BT | Basic testing. A testing phase where the new and/or changed functionality of the finalized software module is tested for the first time. |

| | |
|---|---|
| BVA | Boundary value analysis. A software test case design technique in which test cases are designed to include representatives of boundary values. |
| CIA | Central Intelligence Agency. A civilian intelligence agency of the United States government. |
| CMM | Capability Maturity Model. A model in software engineering of the maturity of the capability of certain business processes. |
| CMMI | Capability Maturity Model Integration. In software engineering and organizational development a process improvement approach that provides organizations with the essential elements for effective process improvement. |
| DNS | Domain Name System. A hierarchical naming system for computers, services, or any resource participating in the Internet. |
| CPU | Central Processing Unit. An electronic circuit that can execute computer programs. |
| ENIQ | Ericsson Network IQ. A performance management application for multi-vendor and multi-technology environments. |
| EP | Equivalence partitioning. A software testing technique that divides the input data of a software unit into partition of data from which test cases can be derived. |
| ETL | Extract, transform, and load. A process of validating, integrating and presenting vastly different sets of data into single coherent set of information. |

| | |
|---|---|
| FTP | File Transfer Protocol. A network protocol used to exchange and manipulate files over a TCP computer network. |
| IBM | International Business Machines Corporation. A multinational computer technology and IT consulting corporation. |
| IEEE | Institute of Electrical and Electronics Engineers. An international non-profit, professional organization for the advancement of technology related to electricity. |
| IT | Information technology. A broad subject concerned with aspects of managing, editing and processing information. |
| JAR | Java Archive. A compressed file which is used to distribute Java classes and associated metadata. |
| JRE | Java Runtime Environment. Combination of the Java Virtual Machine, the Java libraries, and all other components necessary to run Java applications and applets. |
| JVM | Java Virtual Machine. A set of computer software programs and data structures that use a virtual machine model for the execution of other computer programs and scripts |
| RISC | Reduced instruction set computing. A CPU design strategy emphasizing the insight that simplified instructions that "do less" may still provide for higher performance if this simplicity can be utilized to make instructions execute very quickly. |
| ROP | Result Output Period. A period of time during which results are collected. |

| | |
|---|---|
| SASN | Service Aware Support Node. A traffic inspection engine for traffic management in mobile broadband and for charging of mobile data services. |
| SFTP | SSH File Transfer Protocol. A network protocol that provides file transfer and manipulation functionality over reliable data stream. |
| SEI | Carnegie Mellon Software Engineering Institute. A federally funded research and development centre headquartered on the campus of Carnegie Mellon University in Pittsburgh, Pennsylvania, United States. |
| SPARC | Scalable Processor Architecture. A 32- and 64-bit microprocessor architecture from Sun Microsystems that is based on reduced instruction set computing (RISC) |
| SQL | Structured Query Language. A database computer language designed for the retrieval and management of data in relational database management systems, database schema creation and modification, and the database object access management. |
| SSH | Secure Shell. A network protocol that allows data to be exchanged using a secure channel between two networked devices. |
| STDERR | Standard Error. An output stream typically used by computer programs to output error messages or diagnostics. |
| STDOUT | Standard Out. An output stream used by computer programs to write its output. |

| | |
|---|---|
| TCP | Transmission Control Protocol. A set of rules used along with the Internet Protocol (IP) to send data in the form of message units between computers over the Internet |
| Tech Pack | Technology Package. A software module that adds functionality to the Ericsson Network IQ. |
| TMM | Testing Maturity Model. A model used to evaluate the maturity of the testing process. |
| TS | Technical Specification. An explicit set of requirements to be satisfied by a product or service. |
| UT | Unit Testing. A software design and development method where the programmer gains confidence that individual units of source code are fit for use. |
| UTC | Coordinated Universal Time. A time standard based on International Atomic Time with leap seconds added at irregular intervals to compensate the Earth's slowing rotation. |
| x86 | Commercially the most successful instruction set architecture. Usually it implies a binary compatibility with the 32-bit instruction set of the Intel 80386 microprocessor. |
| XML | Extensible Mark-up Language. A general-purpose specification for creating custom mark-up languages. |
| XP | Extreme Programming. An agile software engineering methodology where all software development activities are running simultaneously. |

# 1 INTRODUCTION

Networks get more complicated every day as new nodes are installed regularly by the service providers and new network elements are introduced by the network vendors at an increasing pace. Every network element type has a different set of measurement types and every measurement type contains specific counters. Managing all this information and identifying the possible bottlenecks in the network is almost impossible without a computer and a proper tool.

Ericsson Network IQ (ENIQ) is a performance management application for multi-vendor and multi-technology environments. It collects and processes data for use in performance reporting, resource planning and service assurance. It is a solution that increases and enhances the performance of network assets. It is highly versatile. Its modular Technology Packages make it possible to collect performance data from virtually any network source. It provides end-to-end visibility to personnel accessing reports or queries from the system, all on standard web-based tools.

As the number of network elements, measurement types and counters that ENIQ supports grows constantly, it gets more and more complicated to verify that the data gathered from the network elements is loaded correctly into the database. Because of the time constraints it has been practically impossible to verify that the values loaded into the database were stored in the correct columns. Because of the massive amount of data it was only possible to verify that every measurement type was able to load some data into the database.

To make the testing process easier and more thorough it was decided that an automated testing tool should be developed that could verify the loadings and the verification reports. In the beginning the development was split into two parts so that the tool could be taken into use earlier. Later on the two parts were supposed to be merged together to form a single testing tool.

This thesis was made about the automated load verification part. The goal of this thesis was to automate the Technology Package load testing and make it more thorough than before. Testing in general and software development was studied as part of this thesis to better understand the requirements of the test automation software. With the help of the automation tool developed as part of this thesis it is now possible to verify that every single value has been loaded into the database and it is stored in the correct column.

# 2 SOFTWARE TESTING

## 2.1 History

In the beginning, when computer programs were mainly large-scale scientific or military programs running on mainframe computers, the test cases were usually written on a piece of paper. At the time a finite set of test cases could effectively test the entire system. Tests focused on control flows, data manipulation and computations of complex algorithms. In 1979 Glenford Myers explained in his book, The Art of Software Testing, that "Testing is the process of executing a program or a system with the intent of finding errors". At the time that was probably the best definition of how testing had been done. Testing occurred at the end of the software development cycle and its purpose was to find errors in the finished product. (Dustin, Rashka & Paul 1999, 5; Craig & Jaskiel 2002, 3.)

In the 1980's computers began to spread into people's homes (Timeline of computing, 2009). This led to massive growth of commercial software development. Only the best software companies could survive and their products were widely adopted as standards. The nature of computer programs also changed during this transition. Programs were not just operating in a batch-mode anymore - programs could be called in almost any order. This meant that the number of possible test cases exploded and that testing needed to evolve. So a few years after Myers, in 1983, Bill Hetzel stated in his book, The Complete Guide to Software Testing, that "Testing is any activity aimed at evaluating an attribute of a program or system. Testing is the measurement of software quality." The quality of the software was included as an assessment in the definition of testing by Hetzel. Testing was not just a process to find errors anymore but rather a process to verify the quality of the product. (Dustin, Rashka & Paul 1999, 5, 6; Craig & Jaskiel 2002, 3.)

Although Myers' and Hetzel's definitions were still valid even if their scope was somewhat limited, in 2002 Rick D. Craig and Stefan P. Jaskiel redefined what

testing was in their book Systematic Software Testing. According to Craig and Jaskiel "Testing is a concurrent lifecycle process of engineering, using and maintaining testware in order to measure and improve the quality of the software being tested." The definition of testing does not directly mention finding errors anymore, although it is still valid. Their definition includes not only measuring, but there is also mention about improving the quality of the software. This is known as preventive testing. (Craig & Jaskiel 2002, 3, 4.)

## 2.2 Worst bugs to date

As software is spreading into almost every imaginable place, it becomes more and more important to test it thoroughly. One small and seemingly harmless bug can destroy equipment worth millions of Euros or in the worst case even kill people. The Wired magazine has rated the worst software bugs in the history so far. According to the Wired magazine most of the bugs were caused by poor programming. All the bugs could have been caught with proper testing.

In 1962 Mariner I space probe diverted from its intended path on launch because of a software bug. The mission control had to destroy the rocket. The cause of the accident was discovered to be in the formula that calculates the rocket's trajectory. (History's Worst Software Bugs 2009.)

In 1982 the CIA planted a bug in a Canadian computer system that the Soviets purchased to control the trans-Siberian gas pipeline. The bug was not found by the Soviets and it eventually caused the system to fail, resulting in the largest non-nuclear explosion in the planet's history. (History's Worst Software Bugs 2009.)

Between 1985-87 the Therac-25 radiation therapy device caused the death of at least five patients. The cause for this was that the device was based on an operating system that was put together by a programmer with no formal training. The operating system had a bug called a "race condition", which means that a quick

typist could accidentally configure the device to fire electrons in high power mode straight at the patient. (History's Worst Software Bugs 2009.)

A bug in a software caused AT&T's long distance switches to crash on January 15th 1990. After one of the switches crashed and rebooted all of its neighbours also crashed, and then their neighbours and so on. The reason for this was the message that a neighbouring switch sends out when it has recovered from a crash. Receiving this message caused the switches to crash, which lead to total of 114 switches crashing every six seconds leaving almost 60 000 people without long distance service for nine hours. (History's Worst Software Bugs 2009.)

On June 4th 1996 the Ariane 5 rocket exploded about 40 seconds after lift-off because of a software bug. Some of the code controlling the engine was reused from Ariane 4, but the conversion of large 64-bit values to 16-bit signed integers triggered an overflow condition that resulted in the computer overpowering the rocket's engines, which in turn led to the explosion. (History's Worst Software Bugs 2009.)

In November 2000 a design flaw in a radiation therapy planning software caused a series of accidents due to miscalculation of proper dosage. At least 8 people died and 20 people were seriously injured. The software calculated the dosage based on the order in which data was entered, sometimes delivering a double dose of radiation. The physicians using the software were indicted for murders because they were supposed to verify the calculations by hand. (History's Worst Software Bugs 2009.)

# 3 TESTING TERMINOLOGY

## 3.1 Software defect

Software is designed by people and people will make errors or mistakes. The error might be in the software documentation or in the code. These errors in the software product may lead to a problem as the software does not behave as expected or defined. The errors that have slipped into the software are called defects, bugs or faults. When a defect is executed, it may cause the software product to fail to do what it should, causing a failure. Not all defects will cause failures, which means that software code can contain defects that will stay dormant. (Graham, Van Veenendaal, Evans & Black 2008, 3.)

Errors occur most often when dealing with perplexing technical or business problems, complex business processes, code or infrastructure, changing technologies, or many system interactions. This is because our brains are not designed to handle such complicated or changing tasks and they may not process the information we have correctly. This does not mean that all failures are caused by human errors. Failures can also be caused by environmental conditions. For example strong magnetic or electric fields might cause the hardware or software to fail for various reasons. Someone might even try to cause a failure deliberately. (Graham, Van Veenendaal, Evans & Black 2008, 4.)

The cost of a defect depends highly on when it is found. The cost of fixing a defect grows exponentially towards the end of a software lifecycle. From Figure 1 it can be seen that the cost of fixing a defect is relatively low during the requirements and design phases. The cost quickly rises and in the testing and live use phases the cost is multiple times higher than in the beginning of the lifecycle. (Graham, Van Veenendaal, Evans & Black 2008, 5, 6.)

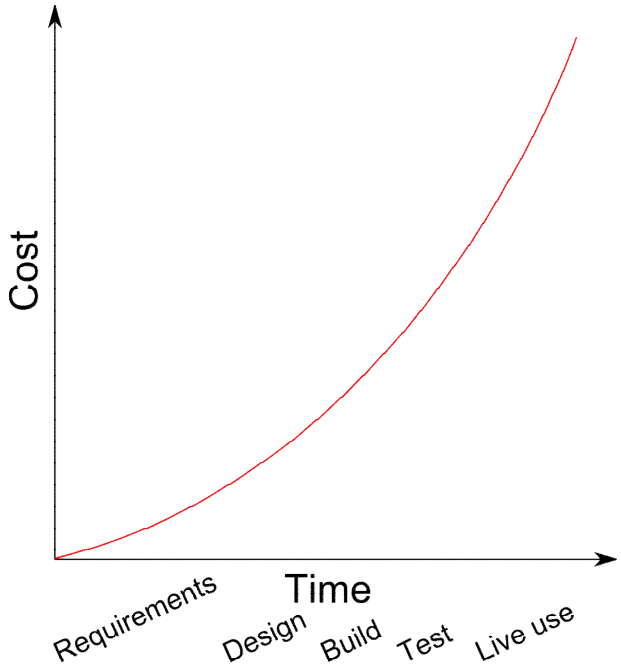



FIGURE 1. Cost of defects (Graham, Van Veenendaal, Evans & Black 2008, 6).

## 3.2 Software quality and quality assurance

There are various definitions of software quality but one of the most common is the IEEE definition. According to the IEEE definition software is a combination of computer programs, procedures, documentation and data necessary for operating the software system. (Galin 2004, 15, 24, 26.)

Software quality assurance always includes all of the components in the IEEE definition of software. The quality of the code is obviously important as the program is the product that the customer ordered. Various documents are needed to ensure the overall quality of the product. Without quality development documentation, efficient cooperation and coordination between the development team members is not possible. The quality of the maintenance documentation is also important as it provides the maintenance team all the required information about the product. This information helps the maintenance team to locate bugs or to add

or change the functionality of the program. The customers' documentation also plays an important role as it describes the available applications and the appropriate methods for their use. Standard test data is an example of essential data necessary to assure the quality of the software. It is used in regression testing to make sure that no undesirable changes in the functionality of the program have occurred or it can be also used to determine what kind of faults can be expected in the software. (Galin 2004, 15, 16.)

## 3.3 Testing phases

When the goal is to develop quality software, the testing cannot be done in one big bang - it needs to be divided into smaller phases. Every test phase targets different types of bugs because there is no single phase that can catch them all. It is impossible to see all the bugs in the beginning when they are cheapest to fix. Each phase has its limitations and benefits which will be examined in the following chapters. (Loveland, Miller, Shannon & Prewitt 2004, 28.)

### 3.3.1 Unit testing

Unit testing (UT) is performed right after the software module is finished, making it the first real test phase the module undergoes. During unit testing the developer tests all new and changed execution paths in the code. The scope of the test is to verify all of the module's inputs, outputs, branches, loops, subroutine inputs and function outputs. If the project is large enough there will be multiple programmers working in parallel to write code and do unit tests on different modules. The modules can also be combined into larger logical components. This is not necessary but in a complex project this will make the next test phase easier. Unit testing is often performed on a virtualized or emulated environment as the native hardware may not be available. (Lewis 2000, 82, 83; Loveland, Miller, Shannon & Prewitt 2004, 29, 30.)

Typical defects found during unit testing include problems with loop termination, internal parameter passing and assignment statements. A major limitation of unit testing is that the module is tested independently and there is no way of knowing how it will perform in a real environment. Drivers and stubs are often required and they are used to simulate the environment around the module. The problem with this is that the drivers and stubs might not work in the same way as the real modules they simulate. Defects that are found during unit testing are cheaper to fix than the ones that are found in the later test phases. (Loveland, Miller, Shannon & Prewitt 2004, 29, 30.)

### 3.3.2 Regression testing

Regression testing is an important part of testing. It is used to detect faults that did not exist in the previous version of the software. Even if some feature has not been changed it could be broken in the newest version. This is because a new or modified component could generate side effects which cause failures in the unmodified part of the code. When new faults are found in the unmodified part of the code the software is said to regress. Before a component is added to the software or an existing component is modified, a baseline version of the working software is put together. This baseline is a version which has been tested before and its faults are known by the development team. (Binder 1999, 755, 756.)

With object-oriented systems regression testing can be run multiple times per day because of the rapid development. This is especially true when the Extreme Programming (XP) approach is used. The Extreme Programming approach requires that a test is developed for every class and it should be rerun every time after the class has been changed. (Binder 1999, 756.)

Regression testing is also used as a first step of integration testing. Rerunning the accumulated test cases when components are added can reveal regression bugs. Regression testing is always an effective part of integration testing and can be used with all integration patterns. When a regression test is executed as a reduced

suite it is also called a smoke test. Smoke tests are used to quickly see if something is broken before running the full regression suite. (Binder 1999, 755, 756, 761.)

### 3.3.3 Integration testing

When all the modules that are to be delivered are unit tested, they are combined in integration testing. Integration testing is usually performed by a separate test team rather than the developers. The focus of integration testing is to verify that the communications between the different software modules works correctly. The white-box testing approach is normally used in integration testing. In integration testing the targeted defects are at a higher level than the defects that can be found during unit testing. Unit testing focused on the internal workings of the module and integration testing targets the module's services and the interfaces that it provides to other modules. (Loveland, Miller, Shannon & Prewitt 2004, 30, 31, 32.)

Integration testing can also be performed on real or virtualized hardware. Virtualized environments can be more versatile than the real hardware because every tester can have their own environments to work with without affecting the others. Integration testing is somewhat limited as its scope is to test single components rather than the whole system. Also, it does not emulate the real load of multiple simultaneous users. Integration testing can be very expensive as the number of testers required to cover all the functions can be high. Test automation tools can help to reduce costs in the long run, although the development costs of such tools in the beginning can be quite high. (Loveland, Miller, Shannon & Prewitt 2004, 30, 31, 32.)

### 3.3.4 System testing

System testing is the first phase where all the pieces of the code are viewed as a single unit. This is also the first time when the testing is done on more realistic loads. The software is usually stress tested during system testing. During stress testing the software is pushed to its limits to ensure stability even in the worst case scenario. As the product is viewed from the customers' perspective the system test team must ensure that the product can be upgraded from one version to another smoothly. System test targets defects such as timing and serialization problems, data integrity and security defects. This is the first time when the product must be tested on native hardware, no virtualization is allowed at this level. There are of course always exceptions to this rule, for example if the product is developed for a virtualized environment. (Loveland, Miller, Shannon & Prewitt 2004, 32, 33, 34.)

Because system testing is limited to a particular product it cannot find cross-product defects. The tools available for debugging in system testing are limited to those that the customer may use. The tools used might be for example logs, trace files or memory dumps and, as a result, the test team might find defects or weaknesses in the tools themselves. System testing is very costly because the hardware needed to perform heavy load and stress tests is expensive. In some environments it is possible to divide the server into multiple partitions, which can help to reduce the costs. (Loveland, Miller, Shannon & Prewitt 2004, 34.)

### 3.3.5 Acceptance testing

Acceptance testing certifies that the software system satisfies the original requirements. This test should be performed after the software has successfully completed system testing. Acceptance testing is performed manually following the acceptance testing plan as closely as possible. Black-box techniques are used to verify the software against its specifications. Acceptance testing continues even if errors are found, unless the error itself prevents continuation. The end users are

responsible for assuring that all relevant functionality has been tested. (Lewis 2000, 84.)

Formal acceptance testing is not always necessary. The customer might be satisfied with the system test or the customer might have been involved in the software development from the very beginning and have been implicitly applying acceptance testing as the software was developed. (Lewis 2000, 84.)

## 3.4 Preventive testing

Preventive testing is the use of techniques and processes that can help to detect and to avoid errors early in the software development cycle when they are easier and cheaper to fix. Preventive testing can also be considered a peer review. It is most effective when the testing is started right after the requirements phase before any code is being written. Reviews can also be done at the code level where it can find potentially problematic design decisions. (Craig & Jaskiel 2002, 4; Dustin 2002, 3; Black 2003, 52.)

Using preventive testing techniques reduces the number of defects that show up during test execution. Even though preventive testing tries to reduce defects it can also point towards solutions. The defects that are found using preventive testing are significantly cheaper to fix than the ones found during the final testing phases. Even if preventive testing is used it does not reduce the need to perform other testing phases, it is just another quality assurance method. (Black 2003, 53.)

Although preventive testing is an old idea, not everyone is using it. According to Craig and Jaskiel, most companies they know are still using some sequential software development process like the Waterfall model. The Waterfall model suggests that once one phase is finished there is no going back, but this is usually not completely true in real development processes. The difficulties arise if you have to back up more than one step, especially in the later development phases. Steve McConnell writes in his book Rapid Development that “Late changes in the Wa-

terfall model are akin to salmon swimming upstream – it isn't impossible, just difficult." (Craig & Jaskiel 2002, 6, 8.)

# 4 TEST AUTOMATION

## 4.1 Why automate testing

Testing is a slow and error prone process if it is done manually. The repetitive nature of the process makes it ideal for automation. Automation is something that must be planned carefully and it must be applied only when a mature manual testing process is already in place. Even if the whole testing process could be automated it does not mean that it should be automated. When automation is applied on a mature testing process, time and money can be saved. When the tests have been automated, the quality of the software also increases as it is possible to run the same tests over and over again with exactly the same inputs. This also means that it is really easy to determine whether the fault has been fixed correctly or not. (Mosley 2002, 4, 5.)

## 4.2 TMM levels and test automation

Testing is divided into five levels according to the maturity of the testing process. The testing Maturity Model (TMM) was created by the Illinois Institute of Technology and it is based on the Capability Maturity Model (CMM), nowadays called Capability Maturity Model Integration (CMMI), which was developed by the Carnegie Mellon Software Engineering Institute (SEI) (Software Engineering Institute 2009). The problem with these models is that they have been designed from the management point of view and offer little or no help to the test automation engineer (Mosley 2002, 2). (Burnstein 2003, 10.)

### 4.2.1 TMM Level 1

At level 1, testing is a chaotic process with no clear test plan. The testing is performed side by side with debugging and the goal of the testing is only to prove that the software works. The final software product is released without quality assurance. Test automation on this level is referred to as “accidental automation”. The test scripts are usually hard to maintain and must be rewritten with each software build. This type of automation can actually increase the testing costs by over 125% compared to manual testing. (Dustin, Rashka & Paul 1999, 17; Burnstein 2003, 12.)

### 4.2.2 TMM Level 2

At level 2, testing is separated from debugging and the tests are usually run after the code is finished. Tests are planned beforehand but they may take place only after coding because the testing process is still immature. The main goal of the testing at this level is to verify that the software meets its specifications. Many quality problems arise at this level because the tests are planned late in the software life cycle. Automation at this level is becoming a planned action. At the second level the test scripts are maintained but the test are not standardized or repeatable. The testing costs can also increase with this type of test automation. (Dustin, Rashka & Paul 1999, 17; Burnstein 2003, 12, 13, 14.)

### 4.2.3 TMM Level 3

At level 3, testing is not just a phase that follows coding. The whole test planning and running is integrated into the software development cycle. Test planning begins at the requirements phase and continues through the software’s life cycle according to the V-model. The test objectives are based on real user and client needs. Automation at this level can be called “intentional automation”. The testing

has become well defined and managed. The automation starts to finally pay off. (Dustin, Rashka & Paul 1999, 18; Burnstein 2003, 14.)

### 4.2.4 TMM Level 4

At level 4 the tests are being measured and quantified. Software products are tested for different quality attributes. These attributes can be for example reliability, usability and maintainability. All the tests are recorded and stored to a test case database and can be then reused in regression testing. Defects are logged and a severity level is assigned to them. At this level automation is referred to as “advanced automation”. When the defect is fixed the fix is tested using the same test cases that were used initially. (Dustin, Rashka & Paul 1999, 18; Burnstein 2003, 14, 15, 16.)

### 4.2.5 TMM Level 5

At level 5, testing has become a well refined process, it is well defined and managed and its cost and effectiveness can be monitored. The tests have to be improved continuously. Automated test tools fully support running and rerunning the test cases. At this level test automation has become even more advanced than in level 4. Test data generation and metrics collection tools such as coverage and frequency analyzers and complexity and size measurement tools are used at this level. Also statistical tools for defect analysis and defect prevention are used. (Dustin, Rashka & Paul 1999, 19; Burnstein 2003, 16.)

# 5 SOFTWARE DEVELOPMENT MODELS

## 5.1 Waterfall model

The waterfall model is one of the earliest software development models designed. In the waterfall model the design and the testing phases are placed on a timeline in sequential fashion. The waterfall model gets its name from the way the model is drawn. The design phases are drawn in a way that the next phase is below the current phase. As the development progresses it flows though the model, hence the name waterfall model. At the top in the waterfall model there are the requirements and design phases and below them the actual implementation. In the bottom end of the waterfall there are the verification and maintenance phases. An example of the Waterfall model can be seen in Figure 2 where the basic steps are illustrated. As testing happens near the end of the development cycle in the waterfall model the defects are detected close to the release date. With the waterfall model it has always been difficult to get the feedback passed up the waterfall. (Black 2003, 19; Graham, Van Veenendaal, Evans & Black 2008, 36.)

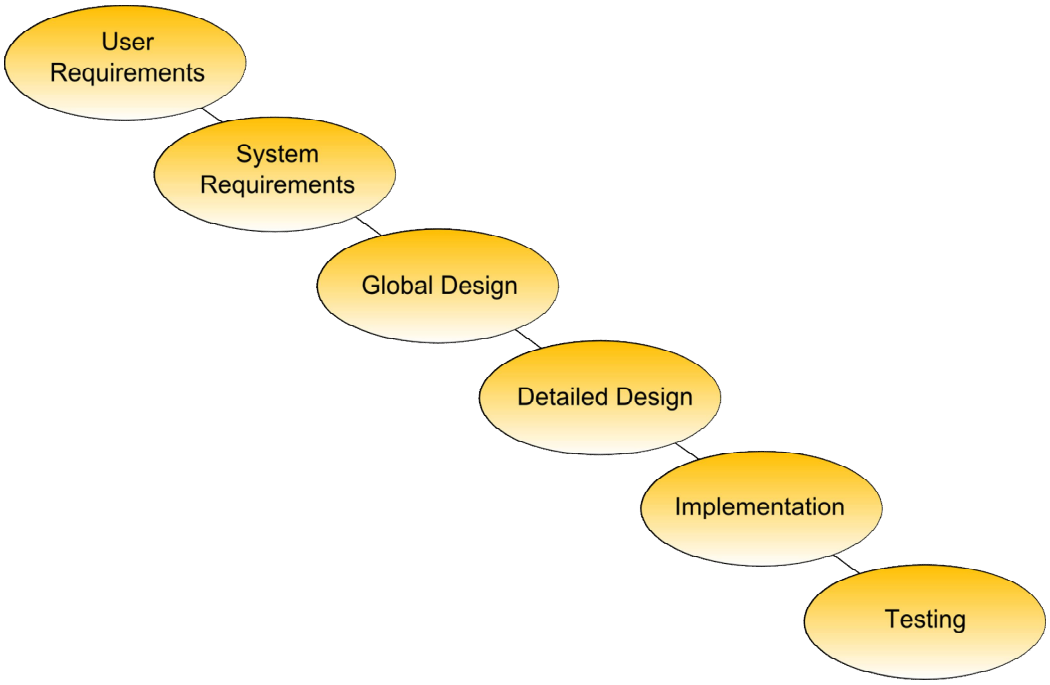



FIGURE 2. The waterfall model (Graham, Van Veenendaal, Evans & Black 2008, 36).

## 5.2 V-model

The V-model is based on the Waterfall model but it combines every design phase from the Waterfall model with a testing phase. Planning for the test phases should start as early as possible, preferably in parallel with the corresponding design phases. There are some variations in V-models but the most common model uses five levels. An example of the V-model can be seen in Figure 3 where the most common levels are depicted. Every design phase relates to a different testing phase as can be seen from the figure. (Black 2003, 19; Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 8.)

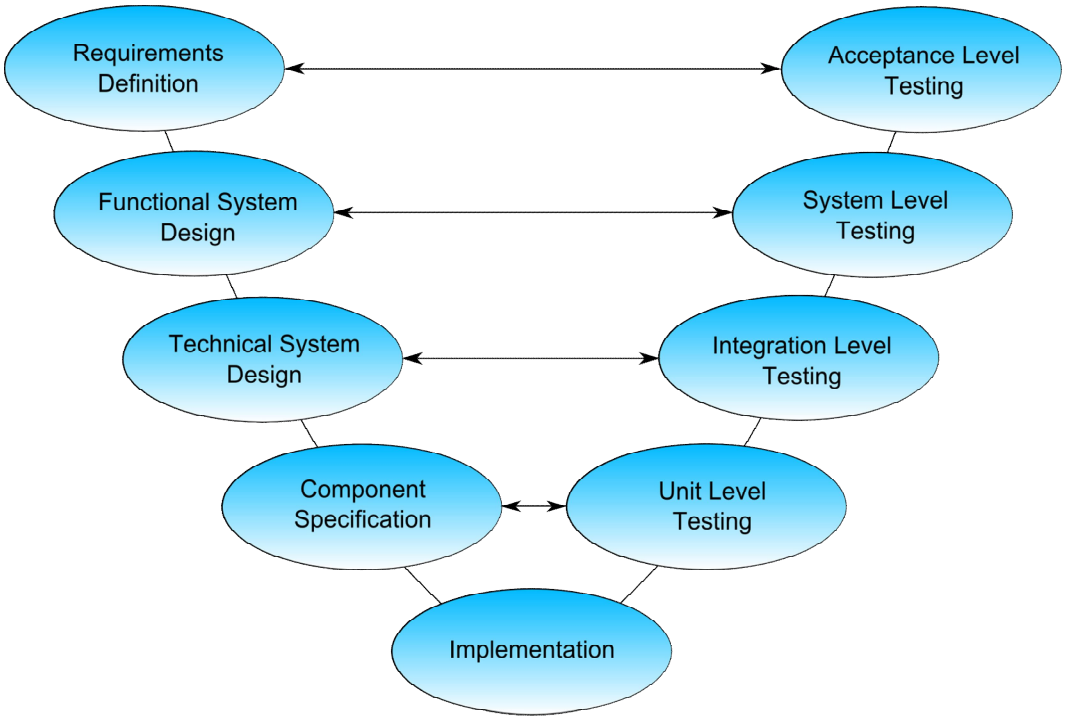



FIGURE 3. The V-model (Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 8).

Component or unit testing is performed at the lowest level in the V-model. The components are always tested against their specifications. Unit tests are usually written and executed by the developer. (Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 8.)

Integration testing tests that communication works between the different software components. Integration testing also verifies that the software can interact with its operating environment. The operating environment can be for example an operating system or the hardware. When all the components are integrated, the system is ready for system testing. (Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 8; Graham, Van Veenendaal, Evans & Black 2008, 37.)

System testing is the first test where the complete system is available for testing. System testing is responsible for verifying that the whole product behaves according to the functional system design. (Baker, Ru Dai, Grabowski, Haugen,

Schieferdecker & Williams 2007, 8; Graham, Van Veenendaal, Evans & Black 2008, 37.)

Acceptance testing is very similar to system testing but it is based only on the users' perspective. Acceptance testing determines whether the product is accepted or not. User needs, requirements and business processes are validated in the process. (Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 9.)

## 5.3 W-model

With the V-model the problem was that the documents on the left hand side of the model did not have a one-to-one relationship with the test phases on the right hand side. The V-model did not take into account the greater value and effectiveness of static tests such as reviews, inspections, static code analysis and so on during the design phases. The W-model is a refined version of the V-model and, like with the V-model, there are different variations of the W-model. The W-model, introduced by Paul Herzlich in 1993 in his book The Politics of Testing, attempts to address the shortcomings in the V-model by introducing a test phase to every development phase. The purpose of the testing phase is to determine whether the corresponding development phase has met its objectives or not. (Gerrard & Thompson 2002, 56, 57, 58; Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 9.)

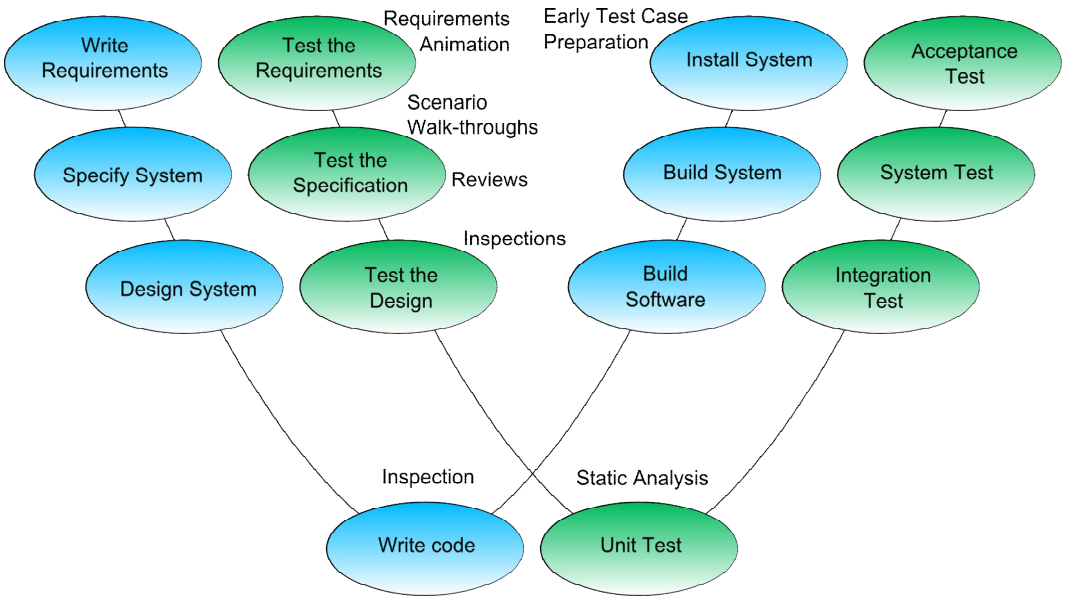



FIGURE 4. The Herzlich's W-model and static test techniques (Gerrard & Thompson 2002, 58).

The Herzlich's W-model is highly adjustable to meet different needs for the development phases even if the phases in use totally differ from the ones in the model. Development activities on the left hand side are always accompanied by the test activities on the right hand side. As can be seen from Figure 4, various static testing techniques can be used with the W-model in the early phases of the software development cycle. Figure 5, on the other hand, demonstrates the different dynamic testing techniques that can be used during the later phases of the W-model software development cycle. (Gerrard & Thompson 2002, 57, 58.)

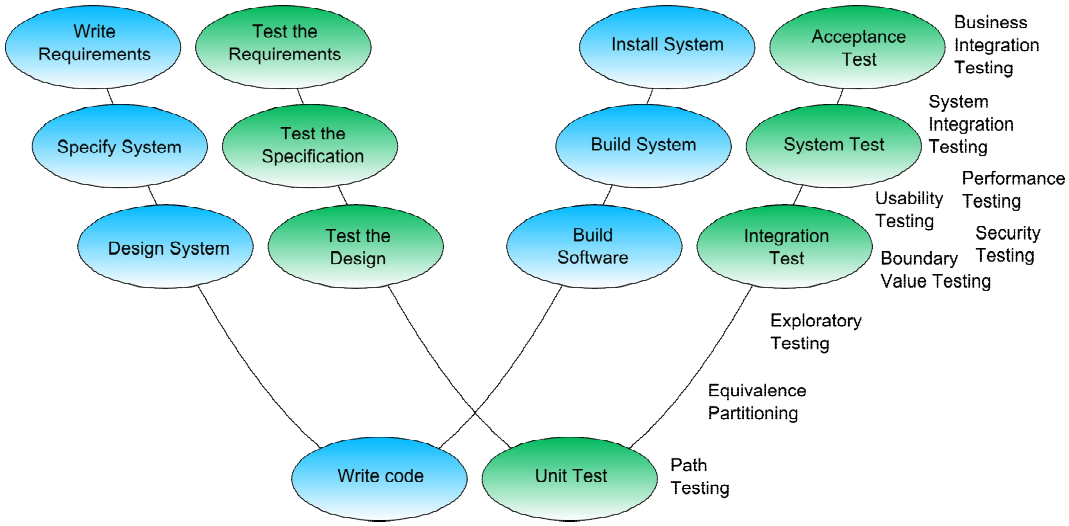



FIGURE 5. The Herzlich's W-model and static test techniques (Gerrard & Thompson 2002, 59).

The W-model can also be used in a real software development process where the number of design phases might not be the same as the number of the testing phases. This was not the case with the earlier models as there was not always the same number of phases in use as in the model. With the Herzlich's W-model for example there might be two or three test phases even though there might be four design phases. With the V-model this would be a problem because according to the V-model's principle, documents from a certain design phase should always be used when defining the test cases to a certain level. Also, none of the design documents should overlap in a testing phase according to the V-model's principle. (Gerrard & Thompson 2002, 56, 59.)

In another W-model, introduced by Dr. Andreas Spillner, the design phases are split into two tasks: a construction task and a corresponding test planning task. The test phases are also split into two tasks that cover the test execution and the debugging. If a fault is found, then the debugging is needed and finally after the required changes have been made to fix the fault, the component has to be tested again from the bottom up. An example of Dr. Spillner's W-model can be seen in Figure 6. Dr. Spillner's W-model emphasizes communication between the differ-

ent design phases and this can be seen as two way arrows in the figure. (Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 9.)

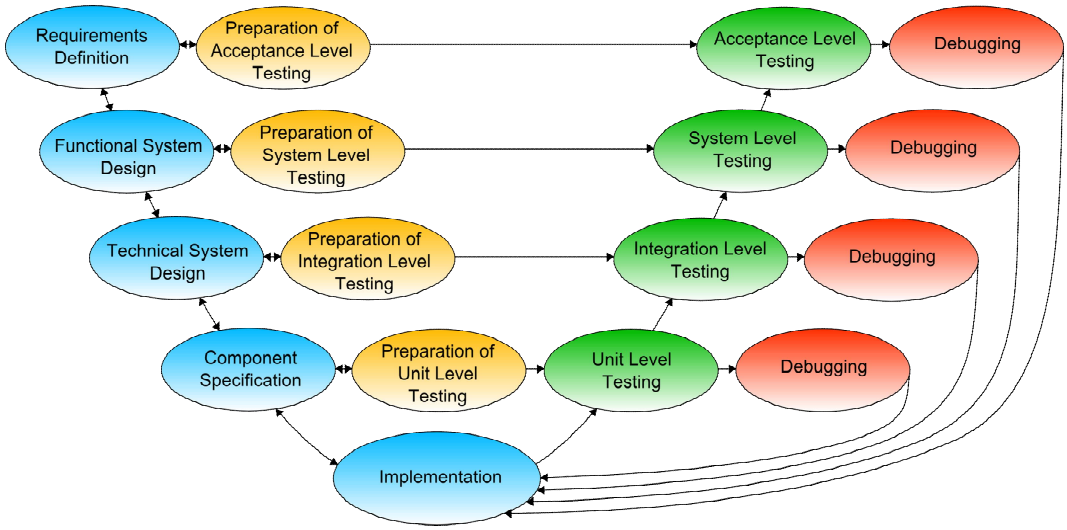



FIGURE 6. The Dr. Spillner's W-model (Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 9).

# 6 TESTING TECHNIQUES

## 6.1 White-box testing

White-box testing, also called structural testing, is a technique used when the internal structure of the component is known. White-box testing is most appropriate on lower levels of testing. Because of its nature, white-box testing is not feasible on higher levels of testing. White-box testing is important because without knowing the internal structure of the component, it is impossible to test all of the ways the component works. This also means that only white-box testing can determine how the component is working. For example a method that should do multiplication on a value might return 4 with an input value of 2. This does not tell whether the multiplication is correctly implemented or not as $2^2$ also equals 4. This is called coincidental correctness and it may slip unnoticed with black-box testing. (Craig & Jaskiel 2002, 160, 161; Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 12.)

Some of the bugs that can be found with white-box testing can also be found with code inspection, which is probably the most effective way of finding logical mistakes. White-box testing requires more skills from the testers than for example black-box testing because in order to perform white-box tests the testers must know how to read the code and the design documentation. (Craig & Jaskiel 2002, 160, 161; Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 12.)

### 6.1.1 Path testing

Path testing is based on flow graphs. Each test case corresponds to a path in the flow graph. Because the number of possible paths could be unlimited there are rules how to define the test cases. Because every statement in the program is expected to be executed, one way to choose test cases is to cover all the statements, although not all commercial testing applications fully support this. This means that there could be dead code that is never reached. Brach coverage is almost identical testing method as statement coverage. Branch coverage targets the nodes where the control flow will divide into two or more possible paths. Even if full statement coverage is reached full branch coverage may not be reached. For full branch coverage every possible path of the program must be tested at least once. (Gao & Wu 2003, 142, 143, 144.)

When a node has multiple conditions it makes sense to test every possible combination of the conditions. It is possible that not all the combinations can be tested because they might be physically impossible, for example in "x > 40 || x < 10" condition x cannot be over 40 and under 10 at the same time. (Gao & Wu 2003, 144, 145.)

Loop statements are the main reason why full path coverage is often impractical, because of the large or infinite number of possible paths. One way to reduce the number of test cases with loop statements is to use boundary testing. With boundary testing the loops can be reduced into only a few possible paths. This means that the loops should be tested with 0, 1, 2, max-1, max and max+1 iterations. (Gao & Wu 2003, 145, 146, 147.)

### 6.1.2 Dataflow testing

When path testing is unfeasible, dataflow testing can be used instead. Dataflow testing focuses on data manipulation, which can be generally divided into two categories: data that defines the value of a variable and data that refers to the value of a variable. Common abnormal scenarios that may cause faults are when a variable is used before it is defined, a variable is defined but never used, and a variable is defined twice before being used. (Gao & Wu 2003, 147.)

As variables can be used in various different contexts, the references to a variable can usually be divided into two categories. The categories are computational use and predicate use. When a variable is used to define the value of another value or it is used to store the output value of some function it is classified as computational use. Predicate use means that the variable is used to determine the Boolean value of a predicate. Test cases should be constructed so that it is possible to test all the references or only one of the reference categories. (Gao & Wu 2003, 147, 148.)

Pointers and array variables increase the complexity of dataflow testing and introduce difficulties to perform a precise dataflow analysis. The cost of dataflow analysis is much higher than that of path testing. (Gao & Wu 2003, 148.)

### 6.1.3 Object-oriented testing

With object-oriented programming the above white-box testing techniques are inadequate as they were originally intended for procedural programming. Object-oriented programming introduces such features as inheritance and polymorphism. With inheritance a subclass may redefine its inherited functions and, because of this, other functions may be affected by the redefined functions. Some of the functions of the parent class might rely on the return value of another function in that same class. Now if this function is redefined in the subclass the other inherited

function that was functioning properly in the parent class might fail. Because of this it is important to test all the inherited functions even if they have already been tested in the parent class. Polymorphism also introduces another problem, because an object may be bound to different classes during the runtime. The things get even more complicated as binding usually happens dynamically. It is possible that randomly selected test cases will miss the faults. (Gao & Wu 2003, 149, 150.)

Other white-box testing techniques can be used with object-oriented programming but because of the nature of the object-oriented programs the adequacy needs to be adjusted. One possibility to test object-oriented programs using the traditional testing approaches is to remodel the program. This means that flow graphs for the classes need to be built. Call graphs can be used to build a flow graph that represents the possible control flows in the program. While this makes it possible to use the traditional white-box testing techniques, it does not address the issues of inheritance and polymorphism. To adequately test object-oriented programs, all the possible bindings and combinations of bindings needs to be tested. (Gao & Wu 2003, 150, 151, 152.)

State-based testing can be used at a high level for black-box testing but it can also be used with object-oriented programs because of features like encapsulation. Encapsulation means that data members and member functions are encapsulated in a class and the data in the class can only be modified through the member functions. These member functions can be used to represent the state transitions of that class. In addition the states defined by the member functions there are two special states in a state diagram; the start state and the end state. The state-based approach can model the behaviour of the program clearly, but obtaining a state diagram from a program is difficult. Generating a state diagram from the source code often yields too many states and the creation of a state diagram based on program specifications cannot be fully automated. (Gao & Wu 2003, 152.)

## 6.2 Black-box testing

Black-box testing is a technique used when the internal structure of the component is not known and is usually used in higher levels of testing. Even when the internal structure is unknown, the interfaces of the component are needed to perform black-box testing. Interfaces define what services the component provides and how. This means that the test cases in black-box testing are partially based on specifications. (Craig & Jaskiel 2002, 159; Gao & Wu 2003, 119, 120, 122; Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 11, 12; Graham, Van Veenendaal, Evans & Black 2008, 87.)

There are various different techniques that can be used with black-box testing. Some of the most common of these techniques, which are described in more detail later on, are equivalence partitioning, boundary value analysis, decision tables and state transition testing. Most of the techniques can be used on all levels of testing but there are a few exceptions. These exceptions can be seen in Table 1. Some of the techniques discussed in this chapter might not be pure black-box techniques but they are generally considered to be black-box techniques. (Craig & Jaskiel 2002, 159; Gao & Wu 2003, 119, 120, 122; Baker, Ru Dai, Grabowski, Haugen, Schieferdecker & Williams 2007, 11, 12; Graham, Van Veenendaal, Evans & Black 2008, 87.)

TABLE 1. Black-box techniques vs. levels of test (Craig & Jaskiel 2002, 162).

| Method | Unit | Integration | System | Acceptance |
|---|---|---|---|---|
| Equivalence Class Partitioning | ✓ | ✓ | ✓ | ✓ |
| Boundary Value Analysis | ✓ | ✓ | ✓ | ✓ |
| Inventories/Trace Matrix | | | ✓ | ✓ |
| Invalid Combinations and Processes | ✓ | ✓ | ✓ | ✓ |
| Decision Table | ✓ | ✓ | ✓ | ✓ |
| Domain Analysis | ✓ | ✓ | ✓ | ✓ |
| State Transition Diagrams | | | ✓ | ✓ |
| Orthogonal Arrays | ✓ | ✓ | ✓ | ✓ |

### 6.2.1 Equivalence partitioning

Equivalence partitioning (EP) is a good all-round black-box technique. It is so basic testing technique that most testers practice it informally even though they may not even realize it. The idea behind the technique is to divide the possible input values into partitions that can be considered the same. If the partitioning is done correctly the system should handle the partitions equivalently. The idea behind EP is that the tester only needs to test one condition from each equivalence partition. This is based on the assumption that if one condition in the partition works then all the values in that partition work. This also means that if one condition in the partition does not work then it is assumed that none of the conditions in that partition work. (Gao & Wu 2003, 127; Graham, Van Veenendaal, Evans & Black 2008, 88.)

All the assumptions that are made during the partitioning process should be documented so that others have a chance to challenge the assumptions. The specification does not always mention all the possible partitions. For example, the specification might say that the password must be at least 8 and at most 20 characters in length. This example would actually have three partitions even if the specification describes only one partition. The invalid partitions must also be included in the partitioning to test the system's behaviour with invalid inputs. Figure 7 illustrates the aforementioned example. The partitions in this example would be; strings that are under 8 characters in length, strings that are between 8 and 20 characters in length and strings that are over 20 characters in length. (Graham, Van Veenendaal, Evans & Black 2008, 88, 89.)

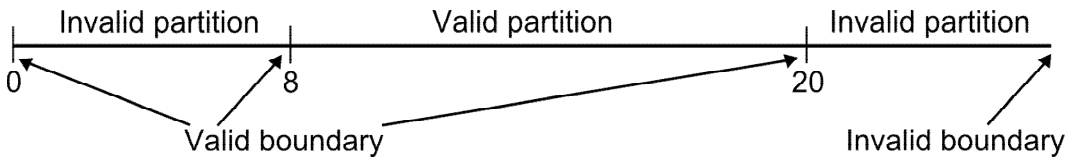



FIGURE 7. Equivalence partitions and their boundaries.

### 6.2.2 Boundary value analysis

Boundary value analysis (BVA) focuses on testing the boundaries between the partitions. The partitions can have both valid and invalid boundaries with open and closed boundaries. A valid boundary is the first or the last valid condition in the partition. Invalid on the other hand is the first or the last invalid condition in the partition. A partition can have either valid or invalid boundaries or a combination of both. In Figure 7 the valid and the invalid boundaries are illustrated for the example that was used to describe the equivalence partitions. This figure has three partitions; the partitions from 0 to 7 and 8 to 20 have valid boundaries, and the partition from 21 onwards has a valid and an invalid boundary. (Graham, Van Veenendaal, Evans & Black 2008, 90.)

A partition has closed boundaries if the minimum and maximum values for that partition are known. An open boundary means that the minimum or maximum value for the partition is unknown. In the example illustrated in Figure 7, one partition has an open boundary. All the other boundaries in the figure are closed boundaries. Even if the partition has an open boundary, its boundary should also be tested. It will be more difficult to test an open boundary than a closed boundary because the boundary can be basically anything. Experienced testers should have an idea what the boundary could be by reading the data type from the specifications. The best way to test an open boundary is actually reading through the specification to find out what the boundary should be specified as. Another way to find the boundary would be investigating the field or data type that is used to store the

value. For example the field in the database could be specified to hold at maximum 5 digit integers. This would mean that the upper boundary value in this case is 99999. This is actually verging on gray-box testing because some of the internal structure is known. (Graham, Van Veenendaal, Evans & Black 2008, 91, 93.)

The program might also receive input through some interface. These interfaces are also a good place to look for partitions and boundaries as the interface might have stricter limits than the field or the data type that is being tested. Finding this kind of defects is usually hard in system testing when the interfaces have been joined together. It is most useful to test the component for these kinds of defects in integration testing. (Graham, Van Veenendaal, Evans & Black 2008, 92, 93.)

There are at least two different boundary value testing methods. The traditional method is to think that the specified limits are the boundaries. This means that three values per boundary are needed to test all the boundary values. According to the traditional method, a valid partition should have closed boundaries. The other method is to think that the boundary is between the valid and invalid values. With this method the number of values per boundary is reduced to two. The traditional method is not as efficient as the other one but both do their job. British Standard 7925-2 for Software Component Testing defines the three value approach. The best method depends on the goals of the testing. If boundary value analysis is combined with equivalence partitioning, testing is slightly more efficient and equally more effective than the three value approach. (Gao & Wu 2003, 131, 132, 133; Graham, Van Veenendaal, Evans & Black 2008, 93, 94.)

### 6.2.3 BVA and EP combined

Boundary value analysis can be combined with equivalence partitioning to form a simple, more thorough testing method. When these testing methods are combined the test cases should be chosen so that one case tests more than one partition or boundary. This way the number of test cases can be reduced and the test coverage stays the same. Only test cases that are thought to pass should be combined into a

single test case. If a test case fails then it is necessary to divide the case into multiple smaller test cases to see what condition has failed. Valid and invalid partitions should not be mixed in the test cases. When invalid partitions are tested the safest way to test them is to have one test condition per test case. This is because the program might only process the first condition, which should in this case fail, and leave the other conditions unprocessed. A good balance between covering too many and too few test conditions is needed. (Gao & Wu 2003, 130, 131, 132, 133; Graham, Van Veenendaal, Evans & Black 2008, 90, 92, 94.)

The reason to do both boundary value analysis and equivalence partitioning is to test whether the whole partition will fail if the boundary values fail. This is also more effective than using only one of them. It can also be much more efficient than running both separately. Testing only the boundary values does not represent the normal values for the field and this does not give much confidence that the program would work under normal conditions in a real environment. What is tested and in what order depends on what is the main goal of testing. The testing could focus on the valid partitions to make sure that the program is ready for release or it could focus on the boundary values if finding defects quickly is important. The most thorough approach is first to test the valid partitions, then the invalid partitions, after that the valid boundaries and finally the invalid boundaries. (Gao & Wu 2003, 130, 131, 132, 133; Graham, Van Veenendaal, Evans & Black 2008, 94, 95.)

### 6.2.4 Decision tables

While equivalence partitioning and boundary value analysis are often applied to specific situations or inputs they are more user interface oriented. EP and BVA cannot be used if a different combination of inputs results in different actions being taken. This is when the decision tables should be used. A decision table is also known as a ‘cause-effect’ table. The decision tables can be used in testing even if they are not used in the specifications although the testing will become easier if the decision tables are used already in the specifications. With decision tables the

testers can explore the effects of different combinations of the possible inputs and how they affect the business logic. (Graham, Van Veenendaal, Evans & Black 2008, 96.)

Testing all the combinations might be impractical or even impossible. Selecting the correct combination of inputs is not trivial and the test may end up being inefficient if a wrong combination of inputs is selected. A large number of combinations should be divided into smaller subsets and the subsets should be tested one at the time. When all the conditions have been identified or a desired combination of conditions is selected, they should be listed in a table and every combination of True and False of those conditions must be tested. The number of combinations to test grows exponentially as the formula for the total number of combinations follows $2^n$, where n is the number of conditions. After all the combinations are listed, the outcome for each combination must be figured out and written in the table. An example of a decision table with conditions and outcomes can be seen in Table 2. If the real result differs from the one that was specified in the table then a defect was found. (Graham, Van Veenendaal, Evans & Black 2008, 96, 97.)

TABLE 2. Decision table for a simple loan calculator (Graham, Van Veenendaal, Evans & Black 2008, 98).

| **Conditions** | **Rule 1** | **Rule 2** | **Rule 3** | **Rule 4** |
|---|---|---|---|---|
| Repayment amount is entered | T | T | F | F |
| Term of loan has been enteterd | T | F | T | F |
| **Actions/Outcomes** | | | | |
| Process loan amount | | Y | | |
| Process term | | | Y | |
| Error message | Y | | | Y |

### 6.2.5 State transition testing

State transition testing can be used when the system or its part can be described in what is called a 'finite state machine'. This means that the system can be only in a number of different states. The system can only go from one state to another by following the rules of the 'machine' and the tests are based on the transitions between these states. An event in one state can only cause one action but the same event in another state can cause a different action and possibly a different end state. This means that the number of states can be greater than the number of events. Figure 8 depicts a state diagram from a simple ATM. The diagram has 7 different states and only 4 different events. The "Pin not OK" event is a good example of an event that causes a different end state depending on when it happens. (Graham, Van Veenendaal, Evans & Black 2008, 100, 101.)

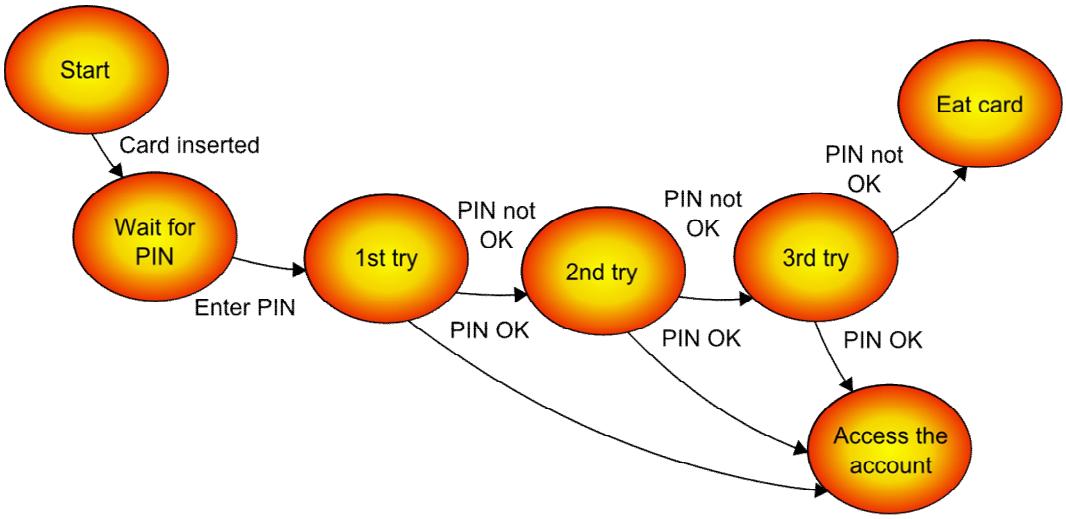



FIGURE 8. State diagram for PIN entry (Graham, Van Veenendaal, Evans & Black 2008, 101).

Different approaches can be taken with state transition testing. Depending on how thorough the test needs to be, either all states or all transitions can be tested. When

the target is to cover all states, the test cases should be planned in a way that minimizes the overlap between state coverage and transition coverage. (Graham, Van Veenendaal, Evans & Black 2008, 102.)

A state chart is a very good tool when state transition testing is used. With a state chart it is easy to see the total number of combinations of states and transitions. A state chart shows both the valid and the invalid transitions. An example of a state table can be seen in Table 3. The valid transitions are the transitions that are documented and that should happen. The invalid transitions on the other hand are the transitions that should not occur under any circumstances because they are physically impossible. Uncertain transitions on the other hand are the transitions which should not happen but they might because they are physically possible. The chart has all the states listed in the left most column and the possible actions are listed across the top row. Possible transitions from one state to another are filled in the cells and those transitions that should be impossible are marked with a dash. Uncertain transitions should be marked with a question mark. The uncertain transitions are a good place to start testing. (Graham, Van Veenendaal, Evans & Black 2008, 102, 103.)

TABLE 3. State table for the PIN example (Graham, Van Veenendaal, Evans & Black 2008, 103).

| | Insert card | Valid PIN | Invalid PIN |
|---|---|---|---|
| S1) Start state | S2 | - | - |
| S2) Wait for PIN | - | S6 | S3 |
| S3) 1st try invalid | - | S6 | S4 |
| S4) 2nd try invalid | - | S6 | S5 |
| S5) 3rd try invalid | - | - | S7 |
| S6) Access account | - | ? | ? |
| S7) Eat card | S1 (for new card) | - | - |

## 6.3 Gray-box testing

Gray-box testing is a combination of black- and white-box testing. Where black-box testing focuses on the program's functionality against the specifications and white-box testing focuses on logic paths, gray-box testing focuses on the program's functionality based on the logic paths. If a function has multiple input parameters, the number of test cases would be factorial of the number of parameters. Without having access to the code the tester would have to write a separate test case for every combination of the parameters. The tester could notice, by talking to the developer or by inspecting the code, that every parameter is independent. This would reduce the number of test cases dramatically. (Lewis 2000, 20, 21.)

# 7 MANUAL TECH PACK TESTING

## 7.1 Basic testing

When a developer has finished creating or updating a Tech Pack it needs to be basic tested to ensure that it works correctly. Basic testing involves different types of tests ranging from documentation related tests to ensuring that every piece of software module is saved into the correct location.

It is quite quick and easy to test that the necessary changes are made into the documentation and that the correct version of the Technology Package is used as a base. Documenting the test results is an important and time consuming part of the testing, which is done during the testing cycle.

### 7.1.1 Definition tests

Basic testing begins with verifying that the Tech Pack is based on the correct version of the functional description. The Tech Pack source files must be named properly and they must be saved into the correct location in the revision control system.

The contents of the source files must also be verified. The sources should have correct version numbers and they must contain all the functionality that is described in the functional description. There is no need to test every item in the source files; only added or modified items need to be verified as all the other items should have been tested when they were implemented.

### 7.1.2 ETL tests

The most time consuming part of the basic testing is the load testing, partly because the loadings should be tested at least for every new or changed measurement type and partly because it involves many stages. To be able to test the loadings, the developer needs to install a data generator tool and configure it correctly for a loading test. Alternatively, the developer could also use existing data that can be transferred with an FTP client into the correct folder on the server. If existing data is used then all the timestamps in the files or in the filenames must be changed. If the timestamps are not changed the data might be ignored because it could be too old. Existing data cannot be used over and over again because it 'wears out'. This means that if the same data is used repeatedly it might not be able to identify new bugs.

When the developer has transferred the files to the server or when the data generator has generated the right amount of data, the developer usually executes loader sets manually from the ENIQ's AdminUI to speed up the process. The loader sets will also execute automatically every fifteen minutes. After a while, when the server has processed all the input files, the developer needs to open the "Show loadings" view from AdminUI to verify that the loadings have succeeded. If something has failed then the cause of the failure needs to be determined and fixed. This procedure needs to be repeated until all the loadings have succeeded.

When the loadings have been executed successfully the developer moves on to test the aggregations. Aggregations are also executed automatically every fifteen minutes but the developer usually starts the aggregation sets manually from the AdminUI. Aggregation testing is quite similar to load testing. The difference is that with aggregations the data has already been loaded into the database. The data is then aggregated based on predefined rules, and it is stored into a different table where the data will be stored for longer periods of time.

### 7.1.3 Universe and report tests

When the aggregations have been successfully tested the developer moves on to test the universes and verification reports. All the new or changed functionality must also be implemented in the universes. If this is not the case then the verification reports will not work as the reports use the universes to fetch the data from the database.

When universes have been tested then all the reports that are new or that are otherwise affected should be opened. By opening the report the developer can verify that the report is able to retrieve data from the database. In case the measurement type supports busy hours then it might have multiple reports.

### 7.1.4 Installation and documentation tests

When all other tests have been executed the installation package of the Tech Pack needs to be tested. The developer needs to verify that the package contains all the necessary files and that everything from the package can be installed or upgraded.

Finally, when all the test cases have been executed the developer needs to finalize the test report and make sure that it is stored with the other necessary documents to the document repository. After everything has been done the Tech Pack is ready for the basic integration testing (BIT).

## 7.2 Basic integration testing

BIT is almost the same as BT but in BIT all the changed modules are installed in the same environment. In BIT the developer should perform all the same tests as in BT. Even when the Tech Pack is working correctly in the BT environment it could fail in the BIT because it might not work correctly with the other changed

modules. This is especially true when the platform modules have changed or some new platform module is introduced.

If some of the tests fail in the BIT environment the developer needs to make the required changes to the Tech Pack. After the changes the Tech Packs should be first tested in the BT environment before the testing can be done again in the BIT environment. This is because the Tech Pack needs to be backwards compatible so it must also work with the previous version of the other modules.

In the end the developers could end up running the tests over and over again for various reasons when they should be already working on their next tasks. This has been a factor that has made the development process slow in the past. With the help of automation, parts of this testing can be performed without manual intervention and during the time the developer is free to work on other tasks.

# 8 AUTOMATING THE TECH PACK BASIC TESTING

## 8.1 Background

There are various commercial automated test tools on the market but those are mainly targeted at user interface testing. IBM's Rational Functional Tester was used as a development tool for this project because in addition to recording user interface related scripts it supports Java programming.

Previously the developer could basically only verify that the different measurement types loaded data but it was practically impossible to verify that the values from the source file actually loaded into the right columns in the database. The idea behind this testing tool was to parse the same files that ENIQ parses and then verify that the values in the database match. The testing tool needed to use different parsers than ENIQ because this increases the possibility of finding faults in the loading process. If the same parsers would have been used, some or all the bugs could have been slipped through unnoticed.

## 8.2 Operating environment

The testing tool was written in Java as Java is widely used within Ericsson. This also makes the future development of the tool easier as there is no need to use unfamiliar programming language. Not all the functionality was implemented into the testing tool itself; instead some external tools were used in conjunction with the tester. The tester uses Plink SSH automation tool for executing commands on the remote server that was running Solaris, Psftp for secure file transfers between the client and the server and 7-Zip command line tool for extracting the compressed files after they have been transferred from the server. The database connection to the Sybase IQ and interface between the tester and the Excel workbook

used to control the tester were provided by external classes provided by Sybase and Apache.

Even though Java programs can be run from different environments this tester needs to run from a Windows environment because of the use of the aforementioned external tools. The tester can be modified to run on UNIX or Linux if needed, but this was not a part of the requirements.

### 8.2.1 Plink & Psftp

Plink and Psftp are utilities of PuTTY terminal emulator application. PuTTY is mainly developed by Simon Tatham. PuTTY is generally considered safe and stable software, which prompted the decision to use its utilities as parts of the tester. As an added bonus the utilities are standalone executables and they can be individually downloaded from the author's web site. This makes upgrading the testers' utilities really easy. One of the most important aspects is that by using PuTTY utilities all connections are encrypted between the server and the workstation that is running the tester.

Plink is a command line variant of PuTTY that can read remote commands from an external file. This was a really important part of the automation because without being able to control the server automatically, the whole project would have not been possible without changing the requirements. Using Plink as a part of the tester was pretty straightforward as it is used by many other programs to perform similar tasks. A fictional Plink remote session can be seen in Figure 9. In the figure Plink is used to tar and compress contents of /var/tmp folder with bzip2. Note that Plink does not write to the console the actual commands sent to the server, it only writes the server's standard output streams to the console. If the program on the server does not output anything then nothing is written to the console.

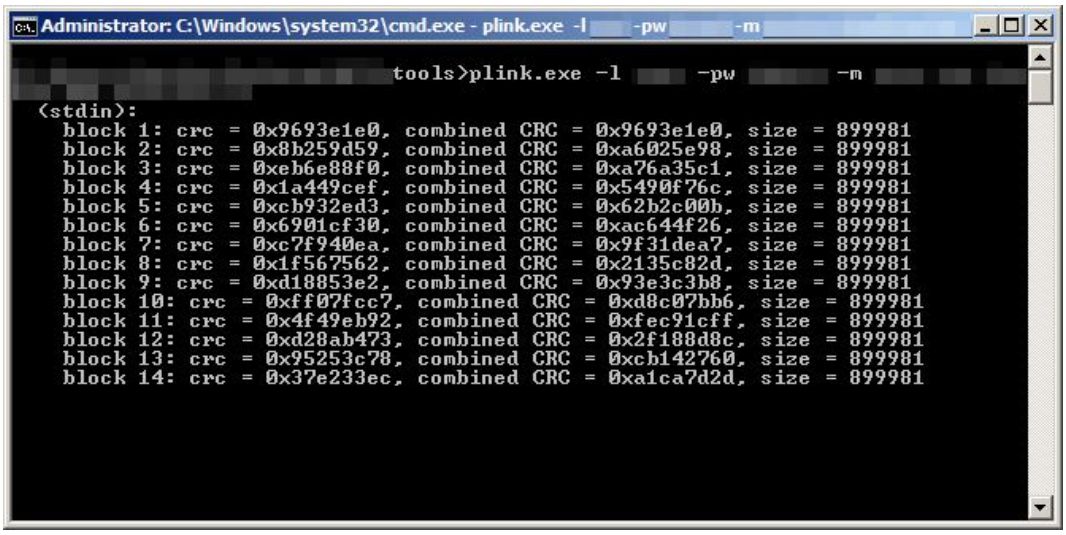



FIGURE 9. An automated Plink remote session example.

Psftp is a command line SFTP client that is based on PuTTY and it can also read commands from an external file. The file transfer functionality was implemented before the automated remote connection functionality and Psftp was not the first utility used for the task. The other tools did not seem very mature or they did not fully conform to the set requirements. So finally when the automated remote connection functionality was implemented with Plink, Psftp was chosen as the file transfer utility. Psftp is invoked similarly to Plink and its console output is also similar to Plink.

### 8.2.2 7-Zip

7-Zip is an open source file archiver developed by Igor Pavlov. 7-Zip was originally designed for Microsoft Windows, but command line ports of the software are now available for most operating systems. When compressing files 7-Zip operates primarily with the 7z archive format. 7-Zip supports extracting data from various different archive formats.

There are two command line versions of the software for Windows. 7z-executable is the version that has support for all the supported archive formats and then there

is the 7za-executable which was used with the tester. 7za-executables support is limited to 7z, ZIP, gzip, bzip2, Z and tar formats. The main reason why 7za-executable was used over the 7z-executable is that 7za-executable can be downloaded individually from the developer's web site and 7z-executable cannot. Because there is no need to download the installer package and extract its contents, updating this utility is much easier with this standalone executable.

The raw files were tarballed from the beginning. This made the file transfer much easier as there was only one file to transfer. 7-Zip was used to extract the tarball from the very beginning. Later it was discovered that the file transfer times were too slow with uncompressed tarballs, so a decision was made to compress the files before the transfer. Because the version of the tar program that is distributed with Solaris does not support compressing the archive, bzip2 was used to compress the files because of its high compression ratio. With compressed tarballs 7-Zip needs to be invoked twice; first it needs to uncompress the compressed archive, after that the tarball needs to be extracted.

## 8.3 Requirements

The tester was supposed to automate the tasks required to generate, load, and test the generated files against the values loaded into the database. It was also required that the tester could be able to test Tech Packs on different servers in one automated run. An Excel workbook was chosen as the control file for the tester as Excel workbooks are usually easy to understand and the information can be divided into multiple sheets. This also makes the management of the information easier because there is only one configuration file for all the Tech Packs.

The tester was supposed to be able to start a data generation on the server and identify differences between the loaded data and raw data files. This process was supposed to be automated as much as possible. The tester is always run from a Windows environment and there was no need to support UNIX or Linux operating systems. If the external tools are changed to their UNIX/Linux variants and the

code is changed so that the generated batch files are in different format it is possible to run the tester also from computers running UNIX or Linux.

The main sheet of the workbook is used to control and configure the workflow of the tester. Every row in the main sheet represents an action. It is not necessary to execute every action defined as there is a column where the action can be disabled. Every type of action has its own columns that are used to configure the execution of that particular action. All the other sheets are named according to the Tech Packs and they are used to specify all the necessary mappings and transformations between the raw data and the database. This information is only used by the testers' parsers and normally there is no need to make changes to them.

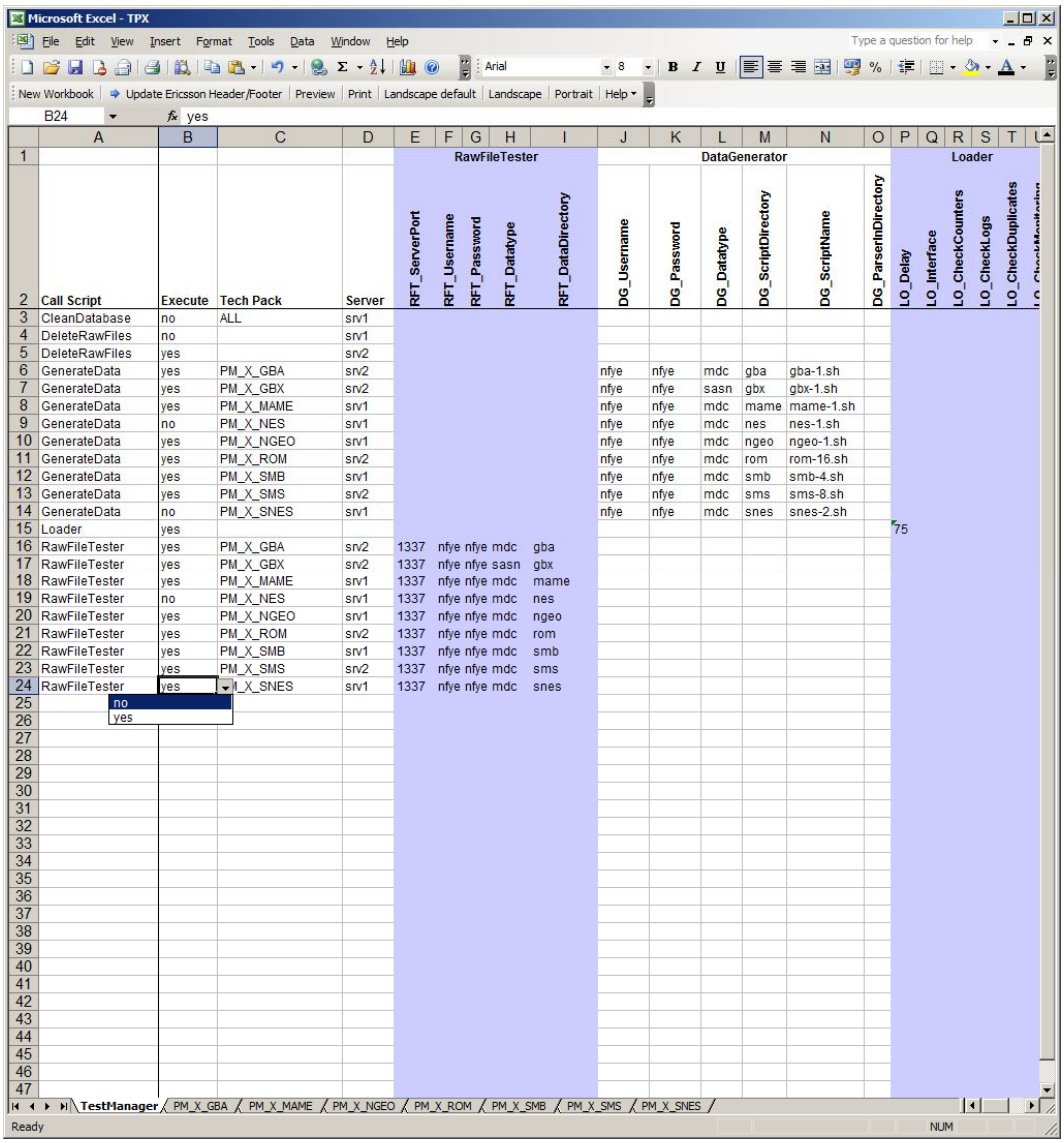



FIGURE 10. Test run configuration.

Figure 10 illustrates the main view of the control file. On the left hand side in the main sheet all the configured actions are listed. The next column to the right defines whether or not the action will be executed during a test run. The next two columns define which Tech Packs are affected and on what server the action is to be run. The rest of the columns are for action specific configurations. The tester was designed so that changing the column order in the control file does not affect its functionality but the names of columns need to be unique.

The tester needed to support various file formats regardless of the file's extension. The file format used with the Tech Pack is specified in the main sheet of the control file. The correct parser version is then chosen automatically based on the contents of the file. During this study parsers were made for XML formatted 3GPP TS 32.401 and 32.435 standards, and also SASN parser for ASCII formatted files was written. Support for ASN.1 formatted files is going to be added in some future version of the tester.

## 8.4 Implementation

The tester was written with as minimal hard coding as possible so a configuration file, which for example contains all the column values in the control file, was written. The configuration file also contains paths for the external tools used and the base paths used on the server. The file is read when the tester is executed and all the values in the file are stored in a HashMap containing String keys and values. The values are then read from this HashMap whenever needed with a get()-method. This makes the code more complex to read but easier to maintain and change if needed. There is usually no need to change anything in this configuration file. The only thing that might have to be changed is the DNS suffix that is concatenated with the server name from the control file.

The tester uses an external tool called EniqSim to generate the test data. EniqSim works by reading sample files and generating similar raw data files. The EniqSim is highly configurable through its setting files and through the sample files. Different configuration files can be used when starting the data generator. Automated Basic Tester (ABT) uses EniqSim with preconfigured scripts and sample files to generate data. These scripts are called through automatically generated Plink batch files. The user has to upload EniqSim and these scripts manually to the server before the testing can begin. In a future version of ABT the EniqSim and all the required files are going to be uploaded automatically to the server.

Because the tester uses an external tool to generate the raw data files rather than using real sample data from the network nodes, there will be some problems running the tests. EniqSim has some bugs that have not yet been fixed and this causes some conflicts with the tester. This is because the ENIQ parsers are externally configured for example to always use the correct timestamp format regardless of the raw data file's timestamp format. ABT's parsers work by following file specifications, and if the timestamp format in the file says that it is local time even though the Tech Pack uses UTC timestamps and ENIQ parses the file correctly, the tester will give errors because the timestamps do not match. This happens only with a few Tech Packs and can be fixed by modifying the raw data file after it is loaded to the local computer.

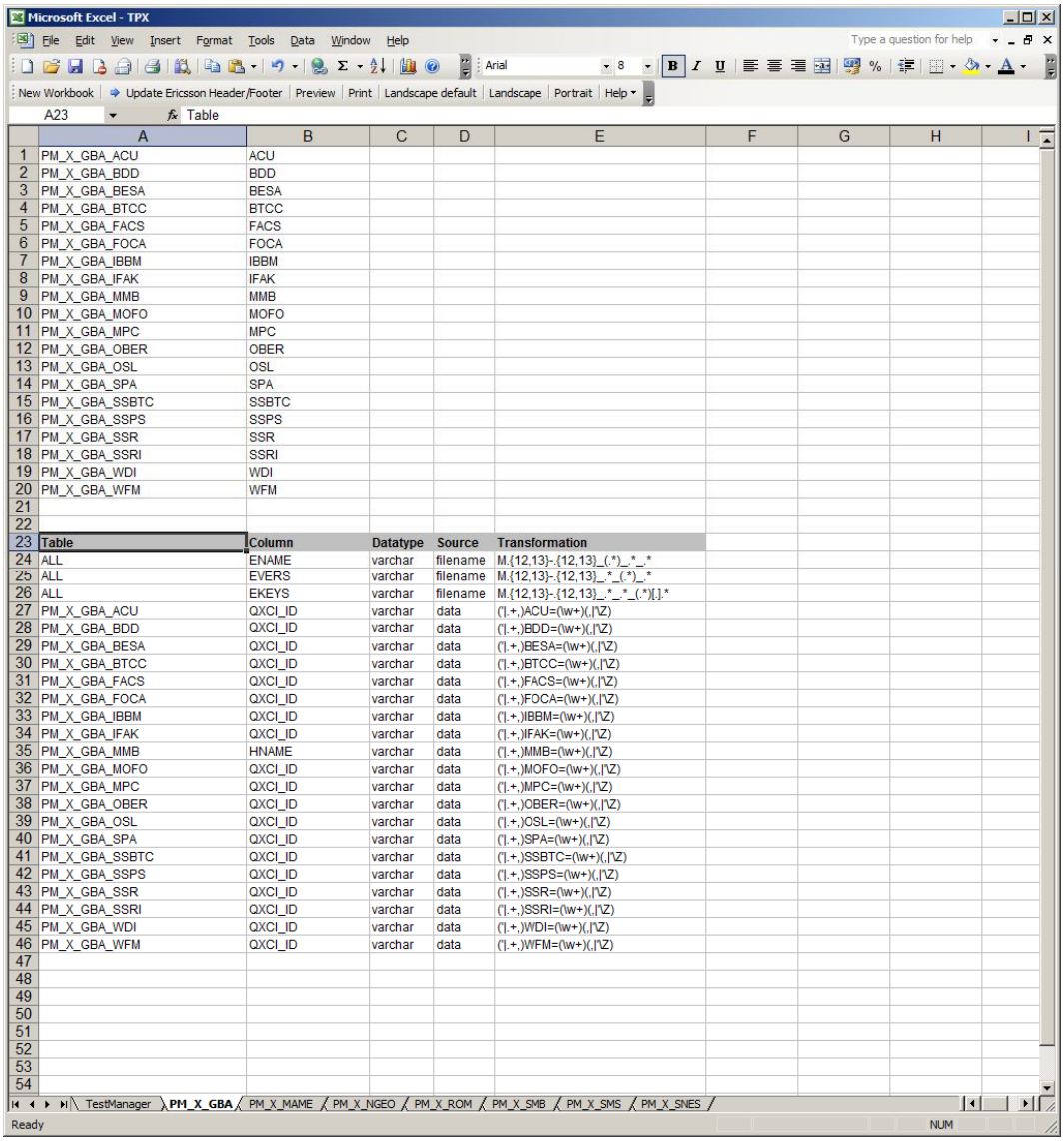
When external programs are executed from JVM the Java program needs to read the standard output streams. Otherwise the external program could stop working. This means that ABT needed to use multiple threads when reading STDOUT and STDERR from the console programs. ABT also uses multiple threads when it is executing the SQL queries. The threads were implemented using functionality that is only available from JRE 6 onwards.

ABT was compiled into a Jar file even though it cannot be executed just by clicking the file. ABT uses more memory than the default JVM when it parses large raw files and also the classpath needs to be defined with the paths to the Jars containing the external classes. A Windows executable was written with AutoIt scripting language to make things easier. This executable can be invoked just like a normal Windows program and it reads the JVM configurations from a file. This file has the JVM maximum memory and classpath preconfigured so that executing the tester is as easy as possible to the user. If needed, the user can increase the maximum JVM memory or change any other JVM options without touching the code.

### 8.4.1 Parsers

The tester's parsers were written to be highly configurable and extendable. This meant that the parsers needed to operate based on an external configuration. The configuration file has a sheet for every Tech Pack where the parser is configured for that particular Tech Pack. The sheet contains mappings from the raw data to the right table in the database and transformations for the data. Transformations are regular expressions that are executed on the specified source and the result is matched with the specified column in the specified table.

An example of a parser configuration sheet can be seen in Figure 11. The figure depicts a regular view of parser configuration. When new measurement types or counters are added to the Tech Pack, those same additions must also be made into the parser's configuration. The transformations are basically the same as the ones that ENIQ uses, but some changes needed to be made before the transformations worked correctly. However, it is recommended to test the transformations with real data and some Java-based tool before changing the parser configuration.




| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | PM_X_GBA_ACU | ACU | | | |
| 2 | PM_X_GBA_BDD | BDD | | | |
| 3 | PM_X_GBA_BESA | BESA | | | |
| 4 | PM_X_GBA_BTCC | BTCC | | | |
| 5 | PM_X_GBA_FACS | FACS | | | |
| 6 | PM_X_GBA_FOCA | FOCA | | | |
| 7 | PM_X_GBA_IBBM | IBBM | | | |
| 8 | PM_X_GBA_IFAK | IFAK | | | |
| 9 | PM_X_GBA_MMB | MMB | | | |
| 10 | PM_X_GBA_MOFO | MOFO | | | |
| 11 | PM_X_GBA_MPC | MPC | | | |
| 12 | PM_X_GBA_OBER | OBER | | | |
| 13 | PM_X_GBA_OSL | OSL | | | |
| 14 | PM_X_GBA_SPA | SPA | | | |
| 15 | PM_X_GBA_SSBTC | SSBTC | | | |
| 16 | PM_X_GBA_SSPS | SSPS | | | |
| 17 | PM_X_GBA_SSR | SSR | | | |
| 18 | PM_X_GBA_SSRI | SSRI | | | |
| 19 | PM_X_GBA_WDI | WDI | | | |
| 20 | PM_X_GBA_WFM | WFM | | | |
| 21 | | | | | |
| 22 | | | | | |
| 23 | **Table** | **Column** | **Datatype** | **Source** | **Transformation** |
| 24 | ALL | ENAME | varchar | filename | M.{12,13}-.{12,13}_(.*)_.*_.* |
| 25 | ALL | EVERS | varchar | filename | M.{12,13}-.{12,13}_.*_(.*)_.* |
| 26 | ALL | EKEYS | varchar | filename | M.{12,13}-.{12,13}_.*_.*_(.*)[.].* |
| 27 | PM_X_GBA_ACU | QXCI_ID | varchar | data | (?i.+,)ACU=(\w+)(,|\Z) |
| 28 | PM_X_GBA_BDD | QXCI_ID | varchar | data | (?i.+,)BDD=(\w+)(,|\Z) |
| 29 | PM_X_GBA_BESA | QXCI_ID | varchar | data | (?i.+,)BESA=(\w+)(,|\Z) |
| 30 | PM_X_GBA_BTCC | QXCI_ID | varchar | data | (?i.+,)BTCC=(\w+)(,|\Z) |
| 31 | PM_X_GBA_FACS | QXCI_ID | varchar | data | (?i.+,)FACS=(\w+)(,|\Z) |
| 32 | PM_X_GBA_FOCA | QXCI_ID | varchar | data | (?i.+,)FOCA=(\w+)(,|\Z) |
| 33 | PM_X_GBA_IBBM | QXCI_ID | varchar | data | (?i.+,)IBBM=(\w+)(,|\Z) |
| 34 | PM_X_GBA_IFAK | QXCI_ID | varchar | data | (?i.+,)IFAK=(\w+)(,|\Z) |
| 35 | PM_X_GBA_MMB | HNAME | varchar | data | (?i.+,)MMB=(\w+)(,|\Z) |
| 36 | PM_X_GBA_MOFO | QXCI_ID | varchar | data | (?i.+,)MOFO=(\w+)(,|\Z) |
| 37 | PM_X_GBA_MPC | QXCI_ID | varchar | data | (?i.+,)MPC=(\w+)(,|\Z) |
| 38 | PM_X_GBA_OBER | QXCI_ID | varchar | data | (?i.+,)OBER=(\w+)(,|\Z) |
| 39 | PM_X_GBA_OSL | QXCI_ID | varchar | data | (?i.+,)OSL=(\w+)(,|\Z) |
| 40 | PM_X_GBA_SPA | QXCI_ID | varchar | data | (?i.+,)SPA=(\w+)(,|\Z) |
| 41 | PM_X_GBA_SSBTC | QXCI_ID | varchar | data | (?i.+,)SSBTC=(\w+)(,|\Z) |
| 42 | PM_X_GBA_SSPS | QXCI_ID | varchar | data | (?i.+,)SSPS=(\w+)(,|\Z) |
| 43 | PM_X_GBA_SSR | QXCI_ID | varchar | data | (?i.+,)SSR=(\w+)(,|\Z) |
| 44 | PM_X_GBA_SSRI | QXCI_ID | varchar | data | (?i.+,)SSRI=(\w+)(,|\Z) |
| 45 | PM_X_GBA_WDI | QXCI_ID | varchar | data | (?i.+,)WDI=(\w+)(,|\Z) |
| 46 | PM_X_GBA_WFM | QXCI_ID | varchar | data | (?i.+,)WFM=(\w+)(,|\Z) |
| 47 | | | | | |
| 48 | | | | | |
| 49 | | | | | |
| 50 | | | | | |
| 51 | | | | | |
| 52 | | | | | |
| 53 | | | | | |
| 54 | | | | | |

FIGURE 11. Parser configuration sheet.

## 8.5 User configurable actions

### 8.5.1 CleanDatabase

CleanDatabase action is used before the testing begins. The purpose of this action is to remove all the unnecessary data from the database. If multiple test runs are

performed and the database is not cleaned between the tests, the server may run out of disk space. Even when only one large Tech Pack is being tested the disk space may run low on the server.

Cleaning the database automatically after the test has run is not recommended. If the tester found loading errors, the user cannot manually run the query which returned the wrong number of results because the database would be empty. However, it is recommended to clean the database always before new data is generated. This way all the loaded data will be accessible after the test run and the servers' disks should always have enough free space.

### 8.5.2 DeleteRawFiles

DeleteRawFiles action is used to remove raw data files from the local computer after the testing has completed successfully. It is also possible to configure this action by changing the values in the appropriate columns in the control file.

Currently there are no safety precautions in the DeleteRawFiles action so the user might accidentally remove something that was not intended. The action should be modified so that it is not possible to navigate upwards in the path. It should also be modified in a way that it checks in the beginning whether or not the server is specified and if it is not then it will just skip the rest of the action.

### 8.5.3 GenerateData

GenerateData action is used to execute EniqSim on the target server and to transfer the generated files to the local computer. After EniqSim has been run the files are tarballed and compressed with bzip2. GenerateData requires that the user has uploaded the EniqSim and the preconfigured scripts manually to the server, otherwise the data generation will fail and there will not be anything to test.

In the future this action will be modified so that the uploading of the EniqSim and the preconfigured scripts will be automated. The action would check whether or not the EniqSim is installed and make decisions based on that. If the EniqSim is not installed then the installation package will be transferred to the server and extracted into the correct folder. In case EniqSim is already installed the action will just start generating the raw data based on the actions configurations.

### 8.5.4 Loader

Originally it was planned that the Loader would start the correct adapters from the AdminUI but Rational Functional Tester had problems recognizing the elements from the AdminUI web site. This was one of the reasons why Loader action became just a configurable delay. After a while it was noticed that it is better that the Loader does not start the adapter sets automatically. This is the way that ENIQ works in real life and it was now possible to test also whether the automatic scheduling of the adapter sets work correctly. If the adapter sets do not work or the testing is started before all the running and queued sets for the Tech Pack in question are executed, the database table will be empty and the tester will give errors.

Another possibility for the Loader action is to use the SSH connection to launch ENIQ adapter sets and then monitor the running and queued sets. This was discussed during the development but at the time it was postponed. When the more important features are already implemented then the loader will be modified so that the operation mode can be configured in the control file.

The delayed Loader action can also be used to start the testing before the testing environment is completely installed. This requires some knowledge of how long the installation will take on the servers' hardware. Installation times vary a lot depending on the servers' architecture. The servers that have SPARC processors need more time to install the Solaris than the ones that have x86 processors. This is probably because of the much lower processor frequency of SPARC.

### 8.5.5 RawFileTester

RawFileTester is the action that handles the actual testing. The action requires that the raw files have been transferred to the workstation and that the server has loaded the values from the raw files to the database. Loader action can be used to make sure that these prerequisites have been met. If the RawFileTester action cannot find files in the directory structure that is parsed from the configuration data then nothing is tested.

When files are found in the correct directory structure the action calls the correct parser one file at the time. When the data is parsed, the action sends the parsed data one record at the time to the SqlGenerator class that constructs all the SQL statements. The constructed SQL statements are then added to a thread queue where they will wait for execution. Because the database systems are designed for multiple simultaneous users, the tester uses a fixed number of threads to execute the queries to speed up the process.

If the executed SQL statement returns more than one row or no rows at all, then the statement is written to the log file with a message telling what the problem with that statement was. The user can then pick up the problematic SQL statements from the log file and execute them manually to see what the problem was. The problem solving is a part that is impractical to automate as there are various possible reasons why something might fail.

Logging is currently implemented directly into the RawFileTester. Logging will be changed in a future version to be totally independent and more versatile. When the logging will be implemented using its own class, logging can be easily changed into something totally different, if needed. The log format will also be changed from a text-based format into an XML-based format, which allows easy manipulation of the data. With a XML formatted log the results can be easily viewed for example with a web browser.

# 9 CONCLUSION

In this thesis the goal was to develop a test automation tool that would make testing more thorough than manual testing and reduce the time needed to perform basic tests. The scope was limited to the loadings part of basic testing as a separate study was made about the automation of the verification report testing. To be able to successfully verify the loadings the column names in the database were also verified in the process.

The test automation tool that was developed proved to be very effective in terms of speed and finding errors. The time needed to verify loadings as thoroughly as possible is now even quicker than or as quick as manually verifying just that the measurement type has loaded something. The time needed to test one result output period (ROP) varies a bit between the test runs because the server and network load is not static. The time needed also depends greatly on the Tech Pack in question as the amount of data gathered during one ROP varies between the Tech Packs.

The tester is mainly configured through an Excel workbook. The workbook contains the required information for the parsers about the Tech Packs and the test run is also configured in the same workbook. A separate file was made for the settings that do not change frequently. These settings include for example DNS suffix, server paths, tool paths and the column headers that are used in the workbook.

In the future support for one more counter type and parsers for at least two data types should be added. At the moment the tester supports single and vector counters, which are the most common ones, MDC data that follows either 3GPP TS 32.401 or 3GPP TS 32.435 standards, and ASCII formatted SASN data.

More actions should also be added in the future to decrease the human effort needed to run the tests. These actions include at least an action that will upload all the necessary files to the server and an action that can detect when the system has been completely installed. When the files can be uploaded automatically, detecting when the installation is complete would speed up the testing even more because at the moment the user can only estimate how long the installation procedure will take. Together these two actions could completely change the way the tester is currently being used.

Logging will be changed into a more readable format as soon as possible. The log format will be some sort of XML file that can be presented in a Web browser or some other tool that supports XML files. The appearance of the XML-based log could be easily changed into something different if needed because of the versatility of the file format. Also, because XML is a widely supported file format the results could be exported into other programs.

All the goals set for the thesis were met successfully. The test automation tool was partly taken into use earlier than required and support for new Tech Packs was added along the way. As a product the tool still needs to be developed further to get the most out of it. At some point in the future the development of the tool will probably be handed over to Ericsson’s site in Ireland.

REFERENCES

Baker, P., Ru Dai, Z., Grabowski, J., Haugen, O., Schieferdecker, I. & Williams, C. 2007. Model-Driven Testing: Using the UML Testing Profile. Springer

Binder, R. 1999. Testing Object-oriented Systems: Models, Patterns, and Tools. Addison-Wesley

Black, R. 2003. Critical Testing Processes: Plan, Prepare, Perform, Perfect. Addison-Wesley

Burnstein, I. 2003. Practical Software Testing: A Process-oriented Approach. Springer

Craig, R. D. & Jaskiel, S. P. 2002. Systematic Software Testing. Artech House

Dustin, E., Rashka, J. & Paul, J. 1999. Automated Software Testing: Introduction, Management, and Performance. Addison-Wesley

Dustin, E. 2002. Effective Software Testing: 50 Specific Ways to Improve Your Testing. Addison-Wesley

Galin, D. 2004. Software Quality Assurance: From Theory to Implementation. Pearson Education

Gao, J. & Wu, Y. 2003. Testing and Quality Assurance for Component-based Software. Artech House

Garfinkel, S. 2005. History's Worst Software Bugs. Condé Nast Publications [referenced 19 March 2009] Available at: http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all

Gerrard, P. & Thompson, N. 2002. Risk-based E-business Testing. Artech House

Graham, D., Van Veenendaal, E., Evans, I. & Black, R. 2008. Foundations of Software Testing: ISTQB Certification. Cengage Learning EMEA

Lewis, W. E. 2000. Software testing and continuous quality improvement. CRC Press

Loveland, S., Miller, G., Shannon, M. & Prewitt, R. 2004. Software Testing Techniques: Finding the Defects that Matter. Charles River Media

Mosley, D. J. 2002. Just Enough Software Test Automation. Prentice Hall PTR

Wikipedia. 2009. Software Engineering Institute. Wikimedia Foundation [referenced 19 February 2009] Available at: http://en.wikipedia.org/wiki/Software_Engineering_Institute

Wikipedia. 2009. Timeline of computing. Wikimedia Foundation [referenced 26 February 2009] Available at: http://en.wikipedia.org/wiki/Timeline_of_computing

# GLOSSARY

This glossary was made by Grove Consultants and it is based on version 6.2 of the Glossary produced by the British Computer Society Specialist Interest Group in Software Testing and it is supplemented by definitions interpreted from the ISEB Foundation Certificate Syllabus version 2.0.

**acceptance testing:** formal testing conducted to enable a user, customer, or other authorized entity to determine whether to accept a system or component. Includes user acceptance testing contract acceptance testing, alpha testing, and beta testing.

**alpha testing:** simulated or actual operational testing at an in-house site not otherwise involved with the software developers.

**beta testing:** operational testing at a site not otherwise involved with the software developers.

**big-bang testing:** integration testing in the small where no incremental testing takes place prior to all the system's components being combined to form the system.

**black box testing:** tests based on the behaviour of the component or system, derived from a specification. Also known as functional testing or behavioural testing.

**bottom-up:** an integration strategy for integration testing in the small where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is included.

**branch:** a conditional transfer of control from any statement to any other statement in a component.

**business process-based testing:** testing based on expected user profiles such as scenarios or use cases. Used in system testing and acceptance testing.

**CAST:** acronym for computer-aided software testing.

**completion criteria:** a criterion for determining when planned testing is complete, defined in terms of a test measurement technique (e.g. coverage), cost, time or faults found (number and/or severity). Also known as exit criteria.

**component testing:** the testing of individual software components. Also known as unit testing, module testing, or program testing.

**contract acceptance testing:** a form of acceptance testing against acceptance criteria defined in a contract.

**coverage:** the degree, expressed as a percentage, to which a specified coverage item (an entity or property used as a basis for testing) has been exercised by a set of tests.

**cyclomatic complexity:** a measure of the complexity of code or a control flow graph, which is equal to the number of decisions plus one.

**decision:** a program point at which the control flow has two or more alternative routes.

**driver:** a specifically written program produced during integration testing in the small to call or invoke a baseline.

**error:** a human action that produces an incorrect result.

**exhaustive testing:** a test case design technique in which the test case suite comprises all combinations of input values and preconditions for component variables.

**expected outcome:** the behaviour predicted by the specification of an object under specified conditions.

**failure:** deviation of the software from its expected delivery or service.

**fault:** a manifestation of an error in software. A fault, if encountered, may cause a failure.

**functional incrementation:** a strategy for combining components in integration testing in the small where they are combined to achieve some minimum capability or to follow a thread of execution of transactions.

**functional requirement:** a requirement that specifies a function that a system or system component must perform. (ANSI/IEEE Std 729-1983, Software Engineering Terminology)

**impact analysis:** assessing the effect of a change to an existing system, usually in maintenance testing, to determine the amount of regression testing to be done.

**incident:** any significant unplanned event that occurs during testing that requires subsequent investigation and/or correction. For example when expected and actual test results are different. Incidents can be raised against documentation as well as code. Incidents are logged when a person other than the author of the product performs the testing.

**incremental testing:** integration testing in the small where system components are integrated into the system one at a time until the entire system is integrated. See also functional incrementation.

**informal review:** a type of review which is undocumented, but useful, cheap and widely used.

**inspection:** a group review quality improvement process for written material. It consists of two aspects; product (document itself) improvement and process improvement (of both document production and inspection). An inspection is led by a trained leader or moderator (not the author), and includes defined roles, metrics, rules, checklists and entry and exit criteria.

**instrumentation:** the insertion of additional code into the program in order to collect information about program behaviour during program execution. Performed by coverage measurement tools in pre-compiler pass.

**integration testing in the large:** testing performed to expose faults in the interfaces and in the interaction between integrated systems.

**integration testing in the small:** testing performed to expose faults in the interfaces and in the interaction between integrated components. Strategies include top-down, bottom-up and functional incrementation.

**isolation testing:** component testing of individual components in isolation from surrounding components, with surrounding components being simulated by stubs.

**LCSAJ:** a Linear Code Sequence And Jump, consisting of the following three items (conventionally identified by line numbers in a source code listing): the start of the linear sequence of executable statements, the end of the linear sequence, and the target line to which control flow is transferred at the end of the linear sequence.

**maintenance testing:** testing changes (fixes or enhancements) to existing systems. May include analysis of the impact of the change to decide what regression testing should be done.

**model office:** an environment for system or user acceptance testing which is as close to field use as possible.

**negative testing:** testing aimed at showing software does not work. Also known as dirty testing.

**non-functional system testing:** testing of system requirements that do not relate to functionality. i.e. performance, usability, etc. Also known as quality attributes.

**oracle assumption:** the assumption that a tester can routinely identify the correct outcome of a test.

**oracle:** a mechanism to produce the expected outcomes to compare with the actual outcomes of the software under test.

**path:** a sequence of executable statements of a component, from an entry point to an exit point. peer review: a type of review which is documented, has defined fault-detection processes, and includes peers and technical experts but no managers. Also known as a technical review.

**precondition:** environmental and state conditions which must be fulfilled before the component can be executed with a particular input value.

**regression testing:** retesting of a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made, and that the modified system still meets its requirements. It is performed whenever the software or its environment is changed.

**reliability:** the probability that software will not cause the failure of a system for a specified time under specified conditions.

**retesting:** running a test more than once.

**review:** a process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users or other interested parties for comment or approval. Types of review include walkthrough, inspection, informal review and technical or peer review.

**static analysis:** analysis of a program carried out without executing the program. Static analysis can provide information about the quality of the software by giving objective measurements of characteristics of the software such as cyclomatic complexity and nesting levels.

**static testing:** testing of an object without execution on a computer. Includes static analysis (done by a software program) and all forms of review.

**stress testing:** testing conducted to evaluate a system or component at or beyond the limits of its specified requirements.

**stub:** a skeletal or special-purpose implementation of a software module, used to develop or test a component that calls or is otherwise dependent on it. Used in integration testing in the small.

**system testing:** the process of testing an integrated system to verify that it meets specified requirements. Covers both functional system testing and non-functional system testing.

**technical review:** a type of review which is documented, has defined fault-detection processes, and includes peers and technical experts but no managers. Also known as peer review.

**test case design technique:** a method used to derive or select test cases.

**test case:** a set of inputs, execution preconditions, and expected outcomes developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. test condition: anything that could be tested.

**test control:** actions taken by a test manager such as re-allocating test resources. This may involve changing the test schedule, test environments, number of testers etc.

**test environment:** a description of the hardware and software environment in which the tests will be run, and any other software with which the software under test interacts when under test including stubs and test drivers.

**test plan:** a record of the test planning process detailing the degree of tester independence, the test environment, the test case design techniques and test measurement techniques to be used, and the rationale for their choice.

**test procedure:** a document providing detailed instructions for the execution of one or more test cases.

**test records:** for each test, an unambiguous record of the identities and versions of the component or system under test, the test specification, and actual outcome.

**test script:** commonly used to refer to the automated test procedure used with a test harness.

**testing:** the process of exercising software to verify that it satisfies specified requirements and to detect faults; the measurement of software quality.

**top-down:** an integration strategy for integration testing in the small where the component at the top of the component hierarchy is tested first, with lower level components being simulated by stubs. Tested components are then used to test lower level components. The process is repeated until the lowest level components have been included.

**user acceptance testing:** part of acceptance testing. Customers or end users perform or are closely involved with the tests, which may be based on business processes or may use a model office.

**validation:** determination of the correctness of the products of software development with respect to the user needs and requirements.

**verification:** the process of evaluating a system or component to determine whether the products of the given development phase satisfy the conditions imposed at the start of that phase. volume testing: testing where the system is subjected to large volumes of data.

**walkthrough:** a type of review of documents such as requirements, designs, tests or code characterized by the author of the document guiding the progression of the walkthrough. Participants are generally peers. Scenarios or dry runs may be used.

**white box testing:** test case selection that is based on an analysis of the internal structure of the component. Also known as structural testing or glass box testing.