*Master Thesis*
*Software Engineering*
*Thesis no: MSE-2002:6*
*March 2002*


# Efficient and Maintainable Test Automation

## A case study of how to achieve efficiency & maintainability of test automation

**Abdifatah Ahmed**
**Magnus Lindhe**

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

This thesis is submitted to the Department of Software Engineering and Computer Science at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 10 weeks of full time studies.

**Contact Information:**
Author(s):
Abdifatah Ahmed
Lindblomsvägen 3B BV
SE – 372 32 RONNEBY

ahmed_abdifatah@hotmail.com

Magnus Lindhe
Stenbocksvägen 8
SE - 372 30 RONNEBY

magnus@lindesign.se

External advisor(s):
Daniel Bergdahl
Ericsson Software Technology AB
+46 457 775 00

Soft Center
The Red Tower, Building VII
SE - 372 25 RONNEBY
SWEDEN

University advisor(s):
Conny Johansson
Department of Software Engineering and Computer Science

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

# Abstract

More and more companies experience problems with maintainability and time-consuming development of automated testing tools. The MPC department at Ericsson Software Technology AB use methods and tools often developed during time pressure that results in time-consuming testing and requires more effort and resources than planned. The tools are also such nature that they are hard to expand, maintain and in some cases they have been thrown out between releases. For this reason, we could identify two major objectives that MPC want to achieve; efficient and maintainable test automation. Efficient test automation is related to mainly how to perform tests with less effort, or in a shorter time. Maintainable test automation aims to keep tests up to date with the software. In order to decide how to achieve these objectives, we decided to investigate which test to automate, what should be improved in the testing process, what techniques to use, and finally whether or not the use of automated testing can reduce the cost of testing. These issues will be discussed in this paper.

**Keywords:** Test Automation, Maintainability, Efficiency, Techniques, and Cost.

# Acknowledgement

We would like to express our gratitude towards Monia Westlund and Bengt Gustavsson for giving us the opportunity to do our research at Ericsson Mobile Positioning Centre in Ronneby.

We would also want to thank and express our appreciation to both our advisors at the university and at EPK, Conny Johansson and Daniel Bergdahl respectively, for their good advice and constructive criticism. We would further like to thank Johan Gardhage for the help and support that he offered to us. Johan was responsible for the automated testing tool that is in use within MPC.

Finally we would like to thank our families and friends that gave us their support and understanding during our master thesis work.

# Abbreviations

| | |
|---|---|
| BT | Basic Test |
| BTS | Base Transmitter Station |
| CGI | Cell Global Identity |
| E-OTD | Enhanced Observed Time Difference |
| ETSI | European Telecommunication Standards Institute (?) |
| FDS | Framework for Flexible Distributed Systems |
| GMLC | Gateway Mobile Location Center |
| GMPC | Gateway Mobile Positioning Centre |
| GPS | Global Positioning System |
| HTTP | Hyper Text Transfer Protocol |
| LCS | Location Services |
| LCSC | Locations Services Client |
| LMU | Location Measurement Unit |
| MLC | Mobile Location Center |
| MPC | Mobile Positioning Centre |
| MPP | Mobile Positioning Protocol |
| MPS | Mobile Positioning System |
| MS | Mobile Station |
| PLMN | Public Land Mobile Network |
| SMLC | Serving Mobile Location Center |
| SMPC | Serving Mobile Positioning Centre |
| SMS | Short Message Service |
| TA | Timing Advance |
| TOA | Time Of Arrival |
| TR | Trouble Report |

# Table of Contents

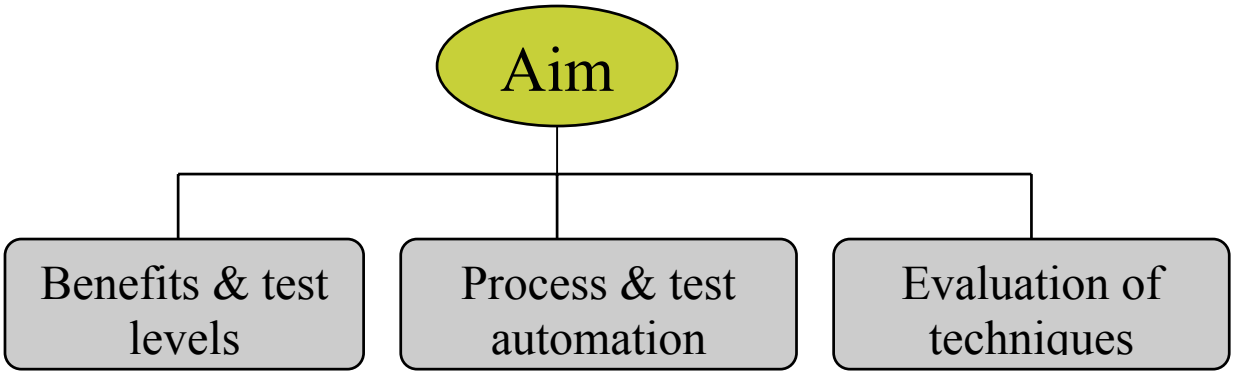
Chapter 1

# Introduction

This paper is the result of the master thesis work performed by Abdifatah Ahmed and Magnus Lindhe. The contents of the paper discuss test automation in relation to testing levels, techniques, process, costs and benefits. The work was carried out at Ericsson Mobile Positioning Centre (MPC) in Ronneby

## Overview of the master thesis

The aim for the thesis is to evaluate automated testing in order to find a solution for how MPC can achieve their test automation objectives; efficient and maintainable test automation.




Test cost

As the picture above shows, the thesis where divided in four major area that relates to our research questions;

1. At what level in the test process (unit, integration or system test) is it beneficial to automate testing?
2. How can the test automation related issues of the test process be improved in order to achieve MPC's test automation objectives?
3. Which automated testing technique(s) is appropriate for MPC?
4. Is it possible to reduce cost of testing with test automation?

The thesis will discuss and answer these questions.

# Disposition

1. **Introduction:** contains an outline of the thesis and a description of our method of working with the master thesis.
2. **Background:** the thesis begins in chapter 2 with a description of domain information related to the background of the MPC department as well as the problem description.
3. **Test automation benefits and test levels:** contains studies on what level in the test process it is beneficial to automate testing and what we believe should be automated in order to achieve efficient and maintainable test automation.
4. **Process related test automation issues:** contains description of the TTM process and what could be improved in order to make the test automation effort more successful, in terms of efficiency and maintainability.
5. **Test automation and cost reduction:** contains a discussion on whether or not test automation can reduce cost for testing. This chapter also discusses the subject of test automation benefits in relation to its costs.
6. **Evaluation of automated testing techniques:** describes different automated testing techniques, their pros & cons and finally a proposal of what techniques are most suited for MPC.

# Method

This section will describe the method we used during the work on the master thesis. The aim of our thesis is to find out how to achieve test automation objectives for MPC. We decided that a case study was the most appropriate research method to use since it "involves investigation of a particular situation, problem, company or group of companies" [4]. In our case we were going to investigate a particular problem, as defined by our aim. We carried out the case study both directly and indirectly by informal interviews and document studies.

These are the MPC documents that was part of our studies:

- Technical description Component FSC-Daily Test
- Thesis proposal Simulators
- Basic Test Plan for PPLocator
- Main Test Plan (Serving Mobile Positioning Centre 5.0)
- Test Plan (Unit, Basic and System Design Test)
- Sub-Process Component Coding + BT (Basic Test)
- http://inside.ericsson.se/ttm/index.html (This is an internal resource which includes several PowerPoint presentations concerning the TTM process)

To establish the aim of our thesis we had a discussion with Daniel Bergdahl who is our supervisor at MPC. From this discussion we found out the problems MPC wanted solutions for and could formulate our aim: evaluate automated testing in order to find a solution for how MPC can achieve their test automation objectives[1].

[1] MPC's test automation objectives will be described in the background chapter.

Together with our supervisor at MPC we decided to use literature and articles as the main source when looking for solutions. We would draw conclusions by combining findings from literature, MPC document studies and interview results with our own opinions. We were not supposed to implement ideas and techniques since we were doing a case study and not action research.

We studied literature in order to give ourselves a brief introduction to the subject of test automation and to determine the scope of our thesis. During the initial literature study, we found out that having a test process that support the automated testing tools was as much important as the tool it self. Therefore, we decided to investigate MPC's current test process with the intention to find out:

- How can the test automation related issues of the test process be improved in order to achieve MPC's test automation objectives

During further discussions with our supervisor and the department manager at MPC, we found some other important questions that was left unanswered:

- Is it possible to reduce cost of testing with test automation?
- At what level in the test process (basic, integration or system test) is it beneficial to automate testing?
- Which automated testing technique(s) is appropriate for MPC?

These four questions above became our research questions and in order to answer them and achieve our aim we decided on the following objectives for our master thesis:

- Study automated testing.
- Study how automated testing is used at MPC.
- Study the MPC development and test processes (TTM) with respect to MPC's automated testing objectives.

To learn how automated testing is used at MPC we first attended a presentation of MPC product and a demonstration of the current automated testing tool. Furthermore, we studied the available documents related to MPC's automated test efforts and discussed the tool and it's design with it's original designer and the designer responsible for the tool at the time of our work with the thesis.

## How we worked with the research questions

To find out how to improve test automation related issues of the test process we looked at the MPC development process called TTM. We did literature and article studies to find issues that were related to MPC's test automation objectives but that we could not find in the TTM process. These issues was described and discussed in relation to the test automation objectives with the intention to raise the awareness of what can be done to achieve the test automation objectives.

To find out if test automation can reduce the cost of testing we focused on articles written by industry professionals as basis for our discussion. We did this because we believe those people have practical experience of the costs and benefits of test automation.

In order to answer the question about test level automation, different ideas and suggestions were gathered from literature and articles. These findings were then critically evaluated so that we finally could make a conclusion on what we believe should be automated in order to achieve efficient and maintainable test automation.

The evaluation of automated testing techniques was done in two steps. The first step was to describe techniques found in literature and articles that was suited for MPC. Then we created selection criteria that single out techniques that help MPC to achieve their test automation objectives. In the case where we did not propose or select a technique for MPC, we recommended guidelines for how to select techniques that could help MPC to achieve their test automation objectives.

## Analysis of data

In order to analyse information that has been collected during literature and MPC studies, we will split up the process of analysing the data in several steps. This will help us to reduce the collected information to a manageable size, so that we can obtain a significance of the analysed information.

- First find out what has been collected
- Identify relationships between MPC current situation, the research question, and MPC test automation objectives
- Identify parts in the MPC documents that indicate efforts done to achieve MPC test automation objectives or obstacles that make it difficult to achieve the objectives.
- Identify literature findings that relates to how to achieve MPC test automation objectives

After doing these steps we were able to understand possible problems and solutions in context to each research question. Then we could make our own conclusions regarding the research questions.

Chapter 2

# Background

## Introduction

The main purpose of the master project is to study the topic of automated testing in order to find a solution for how MPC can achieve their objectives for test automation. In order to do so, we need to provide a background analysis, which will serve as a basis for the requirements of the project.

We will begin by giving a brief overview of mobile positioning in general and the MPC organisation. Following that we will present the problem description.

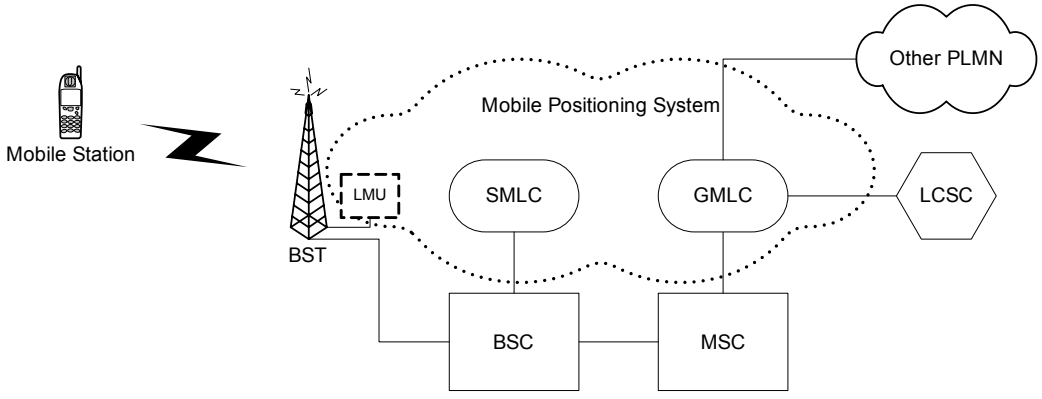
## Background

### Mobile Positioning

This section will explain some contexts in which MPC products operate. If you are not interested in these details or already have an understanding of how mobile positioning works you can skip this section and go straight to the problem description.

#### Overview

Mobile positioning is a technology implemented to provide serviced based on location, called Location Services (LCS). LCS is logically implemented on the GSM structure through the addition of one network node, the Mobile Location Centre (MLC) [15]. The MLC is divided into a gateway part (GMLC) and a serving part (SMLC). All communication with the MPS must go through the gateway, which is responsible for much of the client handling such as authorisation, billing and subscription. A client that communicates with a GMLC is called Location Services Client (LCSC). This can be any kind of application that benefits from mobile positioning such as a fleet management application or perhaps an entertainment application such as a game. A GMLC can also communicate with other Public Land Mobile Network (PLMN) systems. If a positioning request is accepted by the GMLC it will pass it to the serving part, SMLC.



A SMLC is responsible for the actual positioning.

### Timing Advance

Timing Advance (TA) is the simplest way of providing positioning of a Mobile Station (MS). An MS can be any GSM enabled device, most commonly an ordinary mobile phone. The TA is a measurement that can be used to calculate the distance between the MS and its current Base Transceiver Station (BTS). TA is often used together with Cell Global Identity (CGI), which will provide a direction of the MS relative to its current BTS. This method is often referred to as CGI+TA. Today it is possible to position all MS with this method but the accuracy is limited and it is mostly used to assist other positioning systems and as a fallback method.

The area covered by the antennas on a BTS is divided into cells. These can be either sector or omni cells. A sector cell has the form of a slice of pie whereas the omni cell is circular in shape, thus covering the BTS in 360 degrees.. The accuracy of the CGI+TA technique varies depending on the cell network plan. In urban areas with high BTS density and with several sector cells on each BTS the accuracy will be higher than in rural areas where the BTS density is lower and the omni cells are more frequent.

### Time of Arrival

Time of Arrival (TOA) is a technique that makes use of CGI+TA and hyperbolic triangulation to calculate the position of an MS. The method requires that location measurement units (LMU) have been installed at various base stations. At least one out of three BTS used in the triangulation needs to have a LMU installed. The LMU is used to increase the accuracy of the positioning technique in combination with CGI+TA. Today it is possible to position all MS with this technique since it only requires changes in the network. it also gives better accuracy than simply using CGI+TA.

### Enhanced Observed Time Difference

Enhanced Observed Time Difference (E-OTD) is based on TOA and makes use of the observed time difference between several BTS’s. In a synchronised network an MS can calculate the OTD itself without any addition of new hardware and in an unsynchronised network the calculation is assisted by a LMU. Modification to MS software needs to be done to make MS support this kind of positioning. E-OTD is not widely available for consumers as of this date.

### Global Positioning System

Global Positioning System (GPS) is a satellite navigation system that can compute positions in three dimensions worldwide. There are several variations of how the information in the PLMN can be used to assist GPS in the calculation of a MS position. The ETSI standard [15] does not go into details about this but only relying on GPS has several drawbacks in the context of MS positioning.

## Ericsson Mobile Positioning System

The product that Ericsson has implemented using the ETSI standard [15] is called Ericsson Mobile Positioning System (MPS). MPS is developed by Mobile Positioning Centre, which is a department of Ericsson Software Technology AB

and is located in Karlskrona, Kalmar, Ronneby and Malmö. About 100 people are currently working at MPC, distributed over 3 design units, 1 product management unit and 1 test and verification unit. The first MPC product was ready to be used in 1998 and was sold to Telia the following year.

The MPS is developed using a software framework developed internally by MPC called “Framework for Flexible Distributed Systems” (FDS). FDS supports components distributed over several physical servers. Components communicate with each other through a message-based system. The receiver of a component’s messages can be configured in run time. This makes it easy to load a replacement component on a new piece of hardware if a server needs to be shut down for maintenance. The same clever solution can be used to simplify automatic testing as will be described later.

The FDS components that make up the MLC have different areas of responsibilities such as positioning, traffic flow regulation, authorisation, billing, LCSC communication etc.

A free development kit for the latest version of MPS is available for download at Ericsson Mobility World web site (www.mobilityworld.com).

## Automated testing tool that is in use within MPC

### Basic Testing

The basic test environment purpose is to help the designer to test his or hers component. It is made up of the following parts:

- *DailyTest* component
- *JavaTestSender*
- Scripts
- Configuration files

The *DailyTest* component is a FDS framework component like any other that makes up GMPC or SMPC. It provides testing functionality by taking over the roles of other components that the component under test is communicating with. The FDS framework is cleverly made so that it is possible to change where certain messages should be sent. By configuring the component under test to talk to the *DailyTest* component instead of the components it is usually talking to and loading the *DailyTest* component with test scripts it is possible to intercept messages and validate them for correctness. The *JavaTestSender* is a GUI based tool that manages test scripts and the execution of them.

The nature of testing is to perform actions and analyse the results. In the case of basic testing this means that a response message must be generated so the *DailyTest* component can analyse it. Not all messages generate responses that make it harder to test this kind of functionality. It would probably help testing if the design of messages enforced a response for every message. The *DailyTest* component makes it easier to execute tests but verification is often done manually. Even though the *DailyTest* component automates the execution of tests it is still rather cumbersome to configure it and prepare the environment before a test session.

In addition to what has been mentioned in this section, but not really part of basic test, there is a nightly build procedure in place. If a designer’s code breaks the

nightly build that designer will get e-mail with the compiler output and is asked to correct the problem.

## Problem description

In *Automated Software Testing*, Dustin et al [1] address common problems within software organisations that implement automated testing in their software projects. According to Dustin et al [1], "over the last several years test teams have implemented automated testing tools on projects, without having a process or strategy in place describing in detail the steps involved in using the test tool productively". This approach commonly result in the development of test artefacts that are not reusable, which means the test artefacts serves only the current system developed and cannot be applied to a subsequent release of the software application. In the case of incremental software builds and as a result of software changes, these test artefacts need to be recreated repeatedly and must be adjusted several times to accommodate minor changes in the software. This approach increases the testing effort and brings subsequent schedule increases and cost overruns.

Bergdahl [2] describes similar reusability and cost problems as Dustin et al [1] that was experienced within MPC in *Thesis Proposal Simulators*. The document describes that MPC uses methods and tools often developed during time pressure. Using these tools results in time-consuming tests that require a lot of effort and resources. Bergdahl [2] point out that it was hard to maintain and expand the tools, which in turn led to that, the tools were thrown out between MPC versions and could not be reused.

There are two objectives that MPC want to achieve with test automation;

- To make tests more efficient in terms of performing tests with less effort and time.
- To increase the maintainability of test artefacts that is used to automate tests.

# Conclusion

We conclude that the MPC department at Ericsson Software Technology AB experienced problems with time consuming development of automated test tools that in the end was not reusable and had to be thrown away between product versions. Such problems where identified as a common problems within Software Test Automation which means that MPC make the same mistakes as most companies when trying to implement automated testing. Therefore, we will find out:

- How can the test automation related issues of the test process be improved in order to achieve MPC's test automation objectives?
- At what level in the test process (basic, integration or system test) is it beneficial to automate testing?
- Is it possible to reduce cost of testing with test automation?
- Which automated testing technique(s) is appropriate for MPC?

Chapter 3

# Test automation benefits and test levels

## Introduction

The main purpose of this chapter is to answer the question: *at what level in the test process is it beneficial to automate testing?* Beneficial in this context means that testware used to automate tests is efficient and maintainable. We will begin with a description of the testing levels in order to give readers a brief introduction of the basics of testing. Then we will give a description of manual testing and automated testing to clarify the differences and similarities between the two concepts. We will then go deeper into the test automation area in general and identify what to automate so that we will be able to decide what level(s) in the test process that could benefit from test automation. We are not suppose to implement the findings but will propose where in the MPC products to apply these findings.

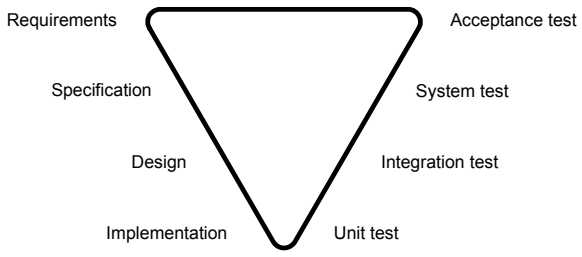
## Overview of testing levels

In this section we will present information related to test process levels to give readers a brief introduction about test levels and why there is a need for such different test levels.

Koomen et al [5] claims that test level is a number of test activities that are organised and directed collectively. Different levels of testing are needed in order to validate whether the program works according to the technical design, whether the application works according to the functional design, and whether the system fulfils the user's needs and wishes. Each test level defines a test strategy to find the most important errors as early and as efficiently as possible where each level addresses a certain number of requirements or functional or technical specifications. Koomen et al [5] have grouped test levels into two categories, low-level tests and high-level tests:

- Low-level tests involve testing separate components of a system, for instance units, programs or modules, individually (unit testing) or in combination (integration testing).
- High-level tests involve testing the whole system where developers test the system integrally (system test), and testing complete products where the system will be offered to the customer for acceptance (acceptance test).

The V Model presented in the *Testing IT* [6], show the relationship between test levels and the software development lifecycle.



# Low level tests

## Unit testing

Unit testing is about testing at the most basic level of the software in order to find errors in program logic. There are two types of unit testing, black box testing (functional) or white box testing (structural).

Black box testing is planned without knowing details of the program design or its implementation. It is usually based on the specification of the program interface, such as procedure and function headers. It also needs to specify the program input and the expected program output.

White box testing is planned with the entire structure of the program design or its implementation. Its aim is testing each aspect of the program logic, driving the test through every single program statement, branch, and path. The required test inputs and the expected output need to be constructed in such a way as to satisfy expected program coverage.

The precise definition of a 'unit' depends on the implementation technology employed when developing a software application. For example, Watkins [6] gave some precise definitions of a 'unit';

- A unit in an application developed using a procedural programming language could be represented by a function or procedure.
- A unit in an application developed using an object-oriented programming language could be represented by a class or an instance of a class, or a method.
- A unit in a visual programming environment or a GUI context could be a window or a collection of related elements of a window, such as a group box.

### Unit testing approach

Some of the example areas for unit testing identified by Watkins [6] are the following:

- Correctness of calculations/manipulations performed by the unit
- Communication between inter operating units
- Low-level performance issues (such as performance bottlenecks observed under repeated invocation of the unit and its functionality)
- Low-level reliability issues (such as memory leaks observed under repeated end extended invocation of the unit and its functionality)

## Integration testing

According to Watkins [6] the objective of integration testing is to determine that the software modules interact together in a correct, stable, and coherent manner prior to system testing. The author [6] also gives the precise definition of a module that again depends on the implementation technology;

- A module in object-oriented programming language could be represented by a collection of objects that perform a well-defined service and that communicate with other component modules via strictly defined interfaces.

- A module in a visual programming environment could be a collection of sub windows that perform a well-defined service and which communicate with via a strictly defined interface.
- A module in a component-based development environment could be a reusable component that performs a well-defined service and that communicates via a strictly defined interface.

Testing is performed against the functional requirements by using the black box testing technique where a test case design demonstrates the correct interfacing and interaction between models, but should avoid any duplication of unit testing effort.

#### Integration testing approach

Some of the example areas for integration testing identified by Watkins [6] are the following:

- Invocation of one module from another inter operating module
- Correct transmission of data between inter operating modules
- Compatibility (that is, checking that the introduction of one module does not have an undesirable impact on the functioning or performance of another module)
- Non-functional issues (such as the reliability of interfaces between modules)

## High level tests

### System testing

During this phase, developers test the system's functionality and stability as well as non-functional requirements such as performance and reliability.

The black box testing technique is used in order to test the high level requirements of the system without considering the implementation details of the component modules.

### Acceptance testing

After the system test has been performed and the encountered defects has been corrected, the system will be offered to the customer for acceptance. During acceptance testing the customer tests the system according to the requirement specification in order to see that the system works correct and is ready for use.

# Testing versus test automation

## Testing

For every system, there are several possible test cases, but yet we are able to test only a very small number of them. These small numbers of test cases are expected to find most of the defects in the software. According to Fewster et al [7] the job of selecting which test case to build and run is an important one and requires the necessary skills to perform the task in the right way. The task of selecting test cases should not be based on random selection but it should be more thoughtful approach if good test cases are to be developed.

The following four attributes has been identified by Fewster et al [7] and these attributes describe the quality of test cases:

- How good/effective is the test cases, in terms of defect detection
- A good test cases should test more then one thing, thereby reducing the total number of test cases required
- How economical a test case is to perform, analyse, and debug
- How evolvable it is, in terms of maintenance effort required on the test case each time the software changes

These four attributes often have to be balanced against one another. So the skill of testing is not only to find defects but test cases should also be designed to avoid excessive cost.

### Objectives for testing

Testing can have many different objectives that will determine how the testing process is organised. For example, if the objective is to find as many defects as possible, then the testing may be directed towards a more complex area of the software. If the objective is to give confidence for end users, then the test may be directed towards the main business scenarios that will be encountered most often in real use. Different organisation will have different objectives for testing or even the same organisation will have different objectives for testing different areas.

## Test automation

According to Fewster et al [7] automated quality is independent of test quality and whether a test is automated or performed manually affects neither how effective tests are in terms of defect detection or testing more then one thing but it effects only how cost effective and evolvable it is. Once implemented an automated test, the cost of running it often will be significant smaller then of the effort to perform it manually and the better approach to automating tests the cheaper it will be to implement them in the long term.

### Objectives for test automation

In order to find a way to assess whether your test automation regime meets your objectives or not, you must first know what your objectives are. You may not need to measure all possible attributes you can think of, but could choose three or four that will give you the most useful information about whether or not you are achieving your objectives. The important thing is to know what your objectives are and to measure attributes that are related to those objectives.

#### Attributes of test automation

The following are attributes of test automation identified by Fewster et al [7]

***Maintainability***
An automation regime that is highly maintainable is one where it is easy to keep tests in step with the software.

***Efficiency***
Efficiency is related to cost and is generally one of the main reasons why people want to automate testing in order to be able to perform their test with less effort, or in a shorter time.

***Reliability***
The reliability of an automated testing regime is related to its ability to give accurate and repeatable results.

***Flexibility***
The flexibility of an automated testing regime is related to the level of extent to which it allows you to work with different subsets of tests. For example, a more flexible regime will allow test cases to be combined in many different ways for different test objectives.

***Usability***
Usability must be considered in terms of the intended users of the regime. For example, a regime may be designed for use by software engineers with certain technical skills, and may need to be easy for those engineers to use. That same regime may not be usable for non-technical people.

***Robustness***
A regime that is more robust will require few or no changes to the automated tests, and will be able to provide useful information even when there are many defects in the software.

***Portability***
The portability of an automated testing regime is related to its ability to run in different environments

## Test automation objectives for MPC

During background analysis we identified objectives that relate to maintainability and efficiency.

The first objective relates to the maintainability of the test automation. Since the existing MPC testing tools are often of such nature that they are hard to expand, adapt and maintain, they have in some cases been thrown out between MPC versions.

The second objective relates to the efficiency of the test automation. MPC want to automate testing in order to be able to perform their test with less effort, or in a shorter time in order to make testing more economical.

### What objectives have been achieved with the Daily-Test tool?

According to Bergdahl [8] the idea behind *DailyTest* tool is "to add formalisation rather than by automation, in the sense of gaining automation as an advantage instead of the other way around". What the author [8] trying to emphasise is that:

- Automation is regarded as the automation of running a complete set of tests in order to facilitate the task of performing tests for designers/testers even under time pressure.
- Establish a formalisation of tests in order to allow designers/testers get a clear view of what functionality is actually there and give a measure of defining development progress to project managers et al. In addition, the Daily-Test tool allows for such formalisation through a well-structured test definition structure.

But the main objectives of test automation for MPC are still remaining and with those objectives in mind, we will find out which tests to automate in order to be able to achieve those objectives.

## Which tests to automate

In this section, we will present different statements made by different authors that relate to which test to automate. These statements will be critically evaluated later in the *Compare and evaluate statements* section. For every set of tests, some will be automatable, others will not. For those that are candidate for automation, you need to decide what test you want to automate in order to achieve your objectives for test automation. Areas where test automation could be beneficial identified by Fewster et al [7] are the following;

- Tests that are straightforward.
- Tests that are difficult to do manually.
- Non-functional requirements
- Regression testing
- Most important tests
- A set of breadth tests (sample each system area overall)
- Tests for the most important functions
- Tests that are easiest to automate
- Tests that will give the quickest payback
- Tests that are run most often

Straightforward tests are for example when you test the functionality of a component where the input and the expected result for that component is known.

Test that is difficult to do manually is for example if you want to perform system test that might require resources that is not available such as, simulating the system condition with thousands of users at the same time.

Testing the performance of the system might be difficult to do it manually while it involves measuring response times under various loading of normal and abnormal system traffic.

Regression test is certainly a candidate for test automation since the objectives with regression testing are to ensure that the system still functioning before you introduce new components or modify the system.

Tests that are more important than others will be run every time something has been changed and others may only need running whenever a particular function changes.

Boehmer et al [12] emphasise the importance of defining a process that will be used to determine what will be automated and propose the following guidelines to follow when setting up criteria for automating targets:

- Automate regression tests: Since regression tests have to be run with every build and will be repeated several times, they make a good candidate for automation.
- Automate tests for stable applications: There is no point in beginning automation on an application that is likely to change in the future. If the application changes, the automated tests must also change, and re-work is never good and might increase testing time.
- Automate time independent tests: Do not automate tests with complex timing issues. If the test is too hard to automate, run the test manually.

- Automate repetitive tests: if a test is a repetitive and boring, they are good candidates for automation.
- Automate tests that have been written: always write test cases before automating them in order to ensure that preparing and writing test case activities are independent of the automation effort. If tests are written by automating, the automation becomes the focus rather than the testing.
- Limit your scope: don't try to automate everything. Achieve small successes then increase your scope as you make progress.

Marick [13] recommends using a decision process based on several questions in order to make a rational decision when deciding what test should be automated. Some of the questions are:

- Automating a specific test and running it once will cost more than simply running it manually once. How much more?
- An automated test has a finite lifetime, during which it must recoup that additional cost. Is this test likely to die sooner or later? What events are likely to end it?
- During its lifetime, how likely is this test to find additional bugs (beyond whatever bugs it found the first time it ran)? How does this uncertain benefit balance against the cost of automation?

Marick [13] describe the third question in detail in terms of what might be lost with automation, how long do automated tests survive, whether or not the automated test will have continued value. Marick [13] identified some other secondary consideration that should be kept in mind when automating test;

- Human ability to notice bugs that where ignored by automation.
- Tools are good when precise checking results while humans might miss it.

In addition to those described above, Marick [13] concluded two things that he believes seem to be broadly true: The first one relates to how to measure the cost of test automation. This is best measured by the number of manual tests it prevents you from running and the bugs it will therefore cause you to miss. The second one is related to whether or not the test automated fulfils the particular purpose designed for the test. According to Marick [13], much of the value of an automated test lies in how well it can do these two things.

I D Hicks et al [10] describe which tests that they could successfully automate:

- Test cases that are perceived to be boring are in fact those exhibiting the greatest simplicity and they should be selected for automation.
- Stress, performance, and capacity testing could only be performed using test automation.

Stress testing is a test that performs with the intention to find out the weaknesses of your system. For example, if you want to load more users than what the system is built for that perform all possible combination of tests until the system fails.

Capacity testing is where you test the system in order to ensure that your system is capable to perform what it is built for. The system fits for its purpose.

## Test automation experience at Microsoft

Angela Smale is an employee at Microsoft and has experience within areas of applications, languages, databases, and operating systems. She has been involved

in activities within many phases of test automation where she describes automating application testing, operating system testing, and many others by using different techniques such as:

- Batch files that was run overnight when they tested DOS
- Capture/playback tools with its own scripting language
- Scripting language that consist of functions to traverse through the user interface UI

These techniques and others with their advantages and disadvantages have been introduced in the book *Software Test Automation* [7]. According to Smale, one should decide areas of which features the users will use 80% of the time or more and these areas will be the top priority for working correctly. Furthermore, she encourages developing Build Verification Tests (BVTs). This is a suite of tests confirming that the basic functionality of a build is still intact. Whenever a build fails this test it is the responsibility of the developer who caused the failure to immediately fix the problem and restart the build. The test team will only install and test a build after the build passes the BVT. The BVT should be run one or two hours after the build of the project is completed. A Development Regression Test (DRT) is a short suite of test that runs 10 to 20 minutes on a private incremental build. This suite of tests covers the basic functionality of the product, and works to prevent developers from checking changes to the code that break primary areas of functionality. This test will be performed just before testers check in changes to the source tree. The DRT can be a subset of the BVT.

Key points for developing automated tests are identified by Smale [7]:

- Automate the most repetitive tasks.
- Automate the tasks that have traditionally found the most bugs.
- Architect the tests so you do not rewrite them for each language
- Modularise your tests for easy maintainability, and reuse them on other projects.
- Keep all your tests in a test case management database.
- Architect the tests to run unattended.

Top-ten list for successful test automation strategies as identified by Smale [7]:

1. Write a detailed test plan before doing anything, be very clear on your automation strategy and get buy-in from your management and peers.
2. Put together a test case management framework, so that each tester is writing to the same standards, and all tests are maintained and accessible.
3. Reduce maintenance – write common functions and modules, and reuse them everywhere.
4. Write meaningful test logs, and generate a summary report for all pass and fail result. Log everything,
5. Have tests run unattended, and capable of recovering from failure.
6. Leverage your tests across multiple languages, platforms, and configurations.
7. Introduce some randomness in your tests.
8. Start small, with tests that are run daily, e.g. build verification tests. Build on success.

9. Measure effectiveness of automation by number and rate of bugs found.
10. Use automation for stress testing – run tests on your product till it fails.

## What should be considered in order to make test automation more maintainable

In this section we will present information related to what attributes that effect the maintainability of test automation. The cost for maintenance is more significant for automated testing than for manual testing. One of the reason is that manual tester is able to implement changes while testing the code manually but for automated tested tool, the tool can not handle changes at runtime simply because the tool has no intelligence.

### Attributes that effect test maintenance

Some attributes that effect test maintenance and alternative solution identified by Fewster et al [7] are the following:

#### Number of test cases

The more tests there are in the test suite the more test there will be to maintain.

An alternative solution to this problem is to consider before adding any test what it will contribute to the test suite as a whole, both in its defect finding capability and its likely maintenance cost. This will help insure that tests are not added for the sake of adding them and even make sure that a consideration of maintenance cost has at least taken place.

Another solution is to go through the automated test suit before product release in order to specially find and remove test cases that are no longer relevant for one reason or another, and test cases that costs more to maintain than the value it provides.

#### Quantity of test data

The more test data there is the more maintenance effort is needed. The effort needed is not only updating test data to reflect new structures, formats, and layouts, but also the task of managing the data takes more effort.

A solution to this kind of problem could be if the task of submitting test cases is made into some form of configuration management system where a formal criterion should be complete in order to limit the total amount of disk space used by individual test cases. This will allow that the management system can automatically check the amount of disk space used and reject any that exceed the limit.

#### Format of test data

The more specialised the format of data used, the more likely it is that specialised tool support will be needed to update the tests.

This could be solved using test format data that is often the most flexible, easily manipulated, and it is also portable across different hardware platforms and system configuration.

### Time to run test cases

Lot of tests rolled into one have inherent inefficiency because of the high coupling and low reusability. The benefits gained in order to avoid multiple set-up and clear-up actions are outweighed by inherent inefficiency of a long test case.

This could be avoided through keeping functional test cases as short and as focused as possible (with reason).

### Debug-ability of test cases

When a test fails, how will I know what went wrong?

If the only information provided by the tool is that test 'failed'. Failure analysis and debugging can be considerable more difficult for an automated test case since manually testers may have a good idea as to what caused a failure.

This problem could be avoided if test cases are designed with debugging in mind by asking 'What would I like to know when this test fails?' The answers to this question may help to structure the way debug information must look like in order to make debugging more helpful.

### Interdependencies between tests

String together a lot of tests such that the outcome from one test case becomes the input to the next might result a failure if one of the test cases fails to produce correct output to the next one.

This could be improved if a few short strings of test cases will be started with in order to see how well they work first, and then expand the number and length as your needs and their effectiveness and efficiency dictate.

### Naming convention

If the number of test cases increases and/or different people become involved, the situation will become chaotic if naming convention where not used.

Adopting some naming conventions right at the start could help to avoid getting into the chaotic situation described above.

### Test documentation

Undocumented or poorly documented test cases will lead to chaotic situation and wastes inordinate amounts of time when maintaining the test cases.

The documentation for test cases must be at the right level and useful. There should be overall documentation giving an overview of the test items as well as annotations in each script to say what the script is doing. Basic strategies and tactics are needed in order to identify those attributes most likely to have the largest impact on test maintenance in your environment. Based on these strategies, something must be done in order to reduce the impact of each one. Fewster et al [7] propose some possible tactics for implementing a strategy for minimizing automated test maintenance costs;

- Define preferred values and standards
- Provide tool support where attributes can be easily measured (such as disk space use and test execution time)
- Keep duplication and redundancy to a minimum
- Provide some form of tool support for the maintenance

# What level in the MPC testing process is it beneficial to automate testing

As the previous section describe about what test to automate, benefits of test automation are not restricted to an specific level within testing process but could be gained in any software testing levels if tests that will be automated will carefully be selected and implemented in the right way. Test automation is a matter of deciding tests that are candidates for automation in order to achieve your objectives for test automation. Therefore, the answer for our research question related to what level to automate is: you can benefit from automating tests in any level of the software test process as long as you carefully select which test to automate in order to achieve your test automation objectives. Further, in order to apply our findings on MPC product, we will identify components in the MPC system that could be candidates for such test automation that we believe could help MPC to achieve efficient and maintainable test automation.

The main focus for the rest of the section will be to compare and evaluate statements related to which tests to automate made by different authors earlier and we will also identify which MPC components that could be candidates for these type of test automation.

## Compare and evaluate statements

In order to be able to compare and evaluate the statements, we will start to categorise different statements made by different authors that relate to what test to automate. Then we will separate those statements that are based on facts from unsubstantiated opinion and even find out whether or not some of the opinions supported by arguments or other authors. Further, we will identify if there are some counter arguments, whether or not we agree statements that made by the authors.

We could identify five categories that are candidates for automation:

1. Tests that are straightforward.
2. Tests that are difficult to do manually.
3. Tests that will be repeated many times.
4. Non-functional requirements.
5. Others that based on rational decision

The reason why we categorize different statements was to make easier for us to evaluate similar statements in the same category at the same time. Our identification of these categories is simply based on to first identify similar statements made by different authors and then create a category for those statements.

### Tests that are straightforward

As Fewster et al [7] stated earlier, tests that are straightforward where the input and the expected output for a stable component is known are candidates for automation. This is true for several reasons:

- According to Hayes [14] the cornerstone of test automation is the premise that the expected application behaviour is known. When this is not the case, it is usually not better to automate.

- Both Hayes [14] and Boehmer et al [12] state that unstable applications whose data is not stable enough to produce consistent results are not good candidates for automation.
- It is quite obvious that this is one of the fundamental issues for automating tests where before tests can be automated, the application/design should be stable enough and both the actual input and the expected outcome are known.

Johan Gardhage is an employee at Ericsson Software Technology AB, is currently responsible for the Daily-Test tool. He pointed out in the early stage of our project that tests that are straightforward are definitely candidates for automation. This was an argument supported by other authors in both literatures and articles, which we also believe that it is true.

Other statements such as, tests that are easiest to automate, tests that will give the quickest payback by Fewster et al [7] seems to be not clear enough to emphasise what authors mean by that. But if tests that are easiest to automate relate to tests that are less complex to design for automation where the inputs and the expected outcomes could be identifies, then this might belong to the straightforward test automation category. Boehmer et al [12] state not to automate tests with complex timing issues there tests might be too hard to automate, and recommend to run them manually. This may relate to how easy or complex a test could be and might be an argument that support the statement made by Fewster [7].

Test that will give the quickest payback in terms of reducing time, research, or may allow performing tests that are difficult to do it manually are of course candidates for automation and could belong different test automation categories described in this section.

### Tests that are difficult to do manually

Some examples area where tests might be difficult to do it manually identified by different authors:

Fewster et al [7] state that simulating the system condition with two hundred users active is good to automate, cause it may be difficult to find 200 volunteers. Hicks et al [10] use the word capacity testing to refer when validating whether or not the system resources could support the forecast customer demands. Hicks [10] also point out that this kind of testing plays an important roll during system testing.

Hicks et al [10] and Smale [7] also emphasise the importance of stress test automation where tests runs on your product till it fails.

Fewster et al [7] state that non-functional testing such testing the performance of the system This is important for two reasons as stated by the authors, it is both difficult to do it manually and might also involve repeating the same test over and over.

There is no doubt about whether or not these tests are candidates for test automation. They all offer a way to perform tests that are difficult to do it manually and also reduce testing time, resource and effort in terms of simulating the system condition with a number of users that might not be available.

### Tests that repeats many times

Many authors emphasised the importance of automating repetitive tests from different point of view. Fewster et al [7] said that reproducing even what one user did previously is not possible if you want to repeat exact timing intervals. Further,

he also point out others such as regression testing and tests that include most important functions are typical repetitive tests:

Boehmer et al [12] also recommend the automation of regression tests since it has to be run with every build and will be repeated several times.

Finally, Bill Boehmer et al [12] and Smale where simple used the term repetitive tests and emphasized that most repetitive test should be automated.

Therefore, we do agree that repetitive tests are candidates for automation for several reasons:

- This is where tools can do the job better than human, thus, executing tests many times in the same way, within same time interval, over and over again.
- This is where tools can be benefits most in order to reduce testing time, resource, and effort when most important tests across many programs are automated.
- Tools are good when precise checking results while humans might miss it.

### Non-functional requirements tests

Non-functional requirement tests that are candidates for test automation such as, performance, maintainability, portability, and others where identified by Fewster et al [7]. The maintainability of tests that are candidates for automation is an essential fact for how profitable automated tests will be and is also an important issue for MPC. Fewster [7], Hicks [10], and Kepple L R [11] describe several attributes, and other factors that might effect test maintenance in the section '*What should be considered in order to make test automation more maintainable*' and we believe that using them as a guidelines will at-least be helpful for those who want to implement maintainable automated tests.

### Others that are based on rational decision

- Marick [13] recommends using a decision process based on several questions that was described earlier.

The first one relates to whether or not automating tests costs less than manually tests. According to Hendrickson [23] this might not work because the use of a method to calculate a precise return on investment ignores some factors. It is difficult to put an accurate dollar amount on the benefit of test automation and this might be almost impossible to quantify the benefits in terms of dollar. We do agree with Hendrickson [23] mainly for one reason. Investment in test automation is rather long term investment than just simple calculating how much does it cost to automate individual tests then doing it manually.

The second one relates to the maintainability of test automation in some degree where answers for this question will help you to think about this issue and we believe that this is relevant question to consider when you are making your first decision towards test automation.

Both Smale [7] and Marick [13] emphasise the importance of automating tasks that have traditionally found the most bugs. We do agree that automating these types of tests will help to find bugs quickly then manually testing which in turn might allow testers to focus on fixing more bugs.

## MPC components that are candidates for automation

The previous section categorizes test automation candidates in five different categories. Some of the candidates offer a way to reduce testing time, resource, and effort where repetitive tests, tests that are hard to do it manually, tests that include the most important functions and others are automated. Other candidates takes into consideration the maintainability issue where a set of breadth tests will be created with the intention to be reused where ever possible and also could be performed repeatedly. Decision-based questions that allow one to consider the lifetime of automated tests also relate to how maintainable tests should be. However, in this section, we will introduce MPC components that could be candidates for these types of test automation.

Some of the criteria for choosing MPC components that are candidates for test automation are the following:

- Base components that will be part of different MPC versions.
- Components that include the most important functions that will be tested every time something has been changed.
- Components that will be tested repeatedly across many application/versions.
- Components that are stable enough to perform a straightforward test automation
- Components that are candidates for regression testing
- Components that include features the users will use most often

The main reason for creating such criteria was to be able to identify components that are candidates for test automation where the automation of tests for these components leads to achieve MPC's test automation objectives.

The creation of these criteria is based on both what has recommended by authors related which test to automate in previous section and also what we believe that should be considered when selecting components for efficient and maintainable test automation. We simple choose to develop criteria from the basis of what has been analysed earlier regarded to what to automate instead of finding other criteria that may or may not exist.

These criteria are not mutually exclusive; they can, and most likely will be used together in order to choose one or several components.

The current MPC version 5.0 has been split into two main products as described in background chapter:

- Serving Mobile Positioning Centre (SMPC)
- Gateway Mobile Positioning Centre (GMPC)

The role of SMPC is to provide a positioning service of mobile stations in the network. GMPC is a gateway between the SMPC and external service providers such as LCS (Location services).

During an interview with Daniel Bergdahl (responsible for System architecture) and Johan Gardhage (responsible for *DailyTest* tool), we used the criteria presented above to identify components that include important tests that are candidate for automation. According to Daniel Bergdahl and Johan Gardhage, there exist components that represent the base of MPC products in both SMPC and GMPC, and therefore these base components will always be part of different MPC versions and include important tests.

We believe that creating test scripts that automate tests for those components could be reused among different MPC versions and will be performed repeatedly many times in order to make sure that the introduction of new module/components does not have an undesirable impact on the basic components (Regression test).

SMPC contains positioning procedure components that are the heart of MPC and some of the most important functions that may tested repeatedly many times across different applications/versions are part of these components. Therefore automating important tests for positioning components will result a greater potential for payback where the tool can perform a long list of repeatable activities over nights or weekends.

## SMPC components

According to Daniel Bergdahl, the basic components for MPC products that make up the SMPC are the following:

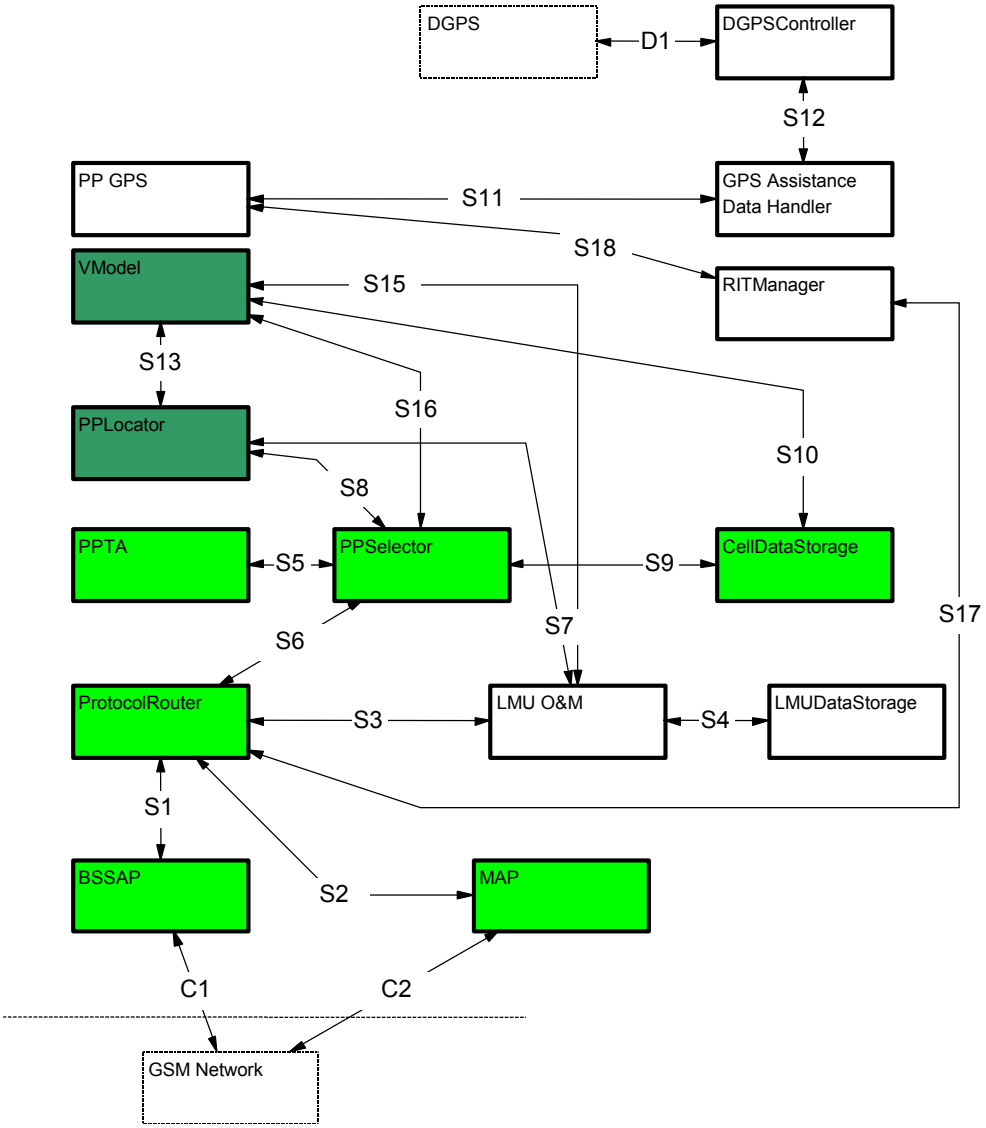
- MAP
- BSSAP
- ProtocolRouter
- PPSelector
- CellDataStorage
- PPTA

Other components identified by Daniel Bergdahl that also could be candidates for automation are:

- PPLocator
- VModel

These two components are delivered by another supplier and therefore will be tested repeatedly and could be candidates for the test automation that we describe early.

The following is MPC component architecture (SMPC 5.0). SMPC components described above that are candidates for automation are coloured.




## GMPC components

According to Johan Gardhage, the most components that make up the GMPC are important for one reason or another. Johan emphasized three different cases based on how often different components are getting involved to perform important, repeatable tasks that include important functions that may be tested repeatedly every time something has been changed.

1. Case 1: used most of the time whenever other components or users getting contact with GMPC
2. Case 2: used often but not as much as the components in case 1.
3. Case 3: used sometimes.

The first four components that belong to case 1:

- Billing

- GeoConv
- RequestMonitor
- MAP

These four components above are candidates for automated testing mainly because they include features where the users will use most often whenever they get a contact with GMPC. These features include important tests that need to be tested repeatedly whenever changes have been introduced to the system.

The following components belongs to case 2:

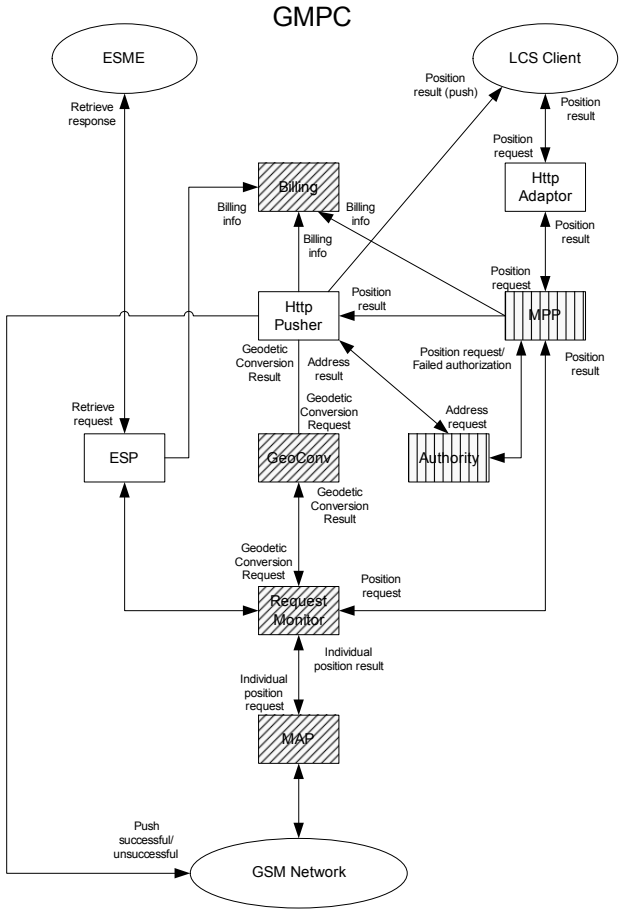
- MPP
- Authority

MPP and Authority components includes important functions that will also be run every time something has been changed and therefore, they are candidates for test automation.

Others that belong to case 3 but are not candidates for test automation so far:

- HTTPServer
- HTTPPusher
- ESP

The following figure shows the internal architecture of GMPC. Components belonging to case one and two are marked differently from each other.



We believe that automating tests for the base components that make up MPC could be reused whenever possible, within the same or different MPC versions which in turn could help MPC with the problems of throwing away tools/scripts between versions. Other important attributes and factors that could have great impact on maintainability of test automation where also described in this chapter, therefore, one should consider these attributes/factors when implementing test automation in order to increase the maintainability of tests that will be automated.

# Conclusion

MPC want to automate testing in order to make tests more economical and evolvable. Therefore, several tests that are candidates for automation identified by different authors where described, compared, and evaluated in this chapter with the intention to decide which test to automate that could help MPC to achieve their objectives with test automation. As described in this chapter, benefits of test automation are not restricted to specific test level but you can benefit from test automation by carefully selecting tests that are candidates for automation in order to achieve your test automation objectives.

For example, automating tests such as stress, system performance, and system capacity testing is related to system testing. Testing the low level performance issue as performance bottlenecks observed under unit testing, or low-level reliability issues such as memory leaks observed under unit testing could also be candidate for automation. Regression test can be used whenever you integrate different parts in individual units or integrating component during integration testing or system integration testing. Therefore, the characteristics of the test itself rather then the testing levels decide whether the test is candidate for automation or could be better perfumed manually. Some of these characteristics are repetitive tests, tests that is difficult to do it manually, tests that include most important functions that will be tested every time changes has been made to the system and others described in this chapter.

During our studies in this particular research question “At what level in the test process (basic, integration, system) is it beneficial to automate testing”, we had a lot of discussions about whether or not this is the right question to ask regarding to benefits of test automation. But since this question was an important one for MPC, we decided to find out what test to automate in order to benefit from automated testing, and then see if benefits of automating tests is dependent on a particular testing level. But as our studies shows, this is not the case, it’s not testing levels that decide the benefits of tests automation but the characteristics of the test. Therefore, the answer to what level to automate tests is: you can benefit from automating tests in any level of the software test process as long as you identify and select tests in all levels that should be automated in order to achieve your test automation objectives.

Furthermore, in order to apply our findings on MPC products, we hade a discussion with our supervisor at MPC where we finally agreed upon to identify components in the MPC system that could be candidates for tests automation by using criteria that is based on both what has recommended by authors related which test to automate in previous section and also what we believe that should be considered when selecting components for efficient and maintainable test automation.

This was not really part of answering our research question but we decided to at least identify where to apply our findings during literature studies. We were not supposed to go more detail into the structure of these components or propose implementation suggestions for our findings, because these were out of the scope of this chapter. Component identification was performed during an interview with our supervisor at Ericsson Bergdahl, and Gardhage who was responsible for the test tool. By using selection criteria that was developed during our studies, we were able to sort out which components that fulfil these criteria.

Other important attributes and factors that could have great impact on maintainability of test automation and their solution where also described in this chapter, therefore, one should consider these attributes when implementing test automation in order to increase the maintainability of tests that will be automated.

Automating repeatable tests, tests that are difficult to do it manually, regression test, and others described in this chapter will lead to reduce testing time, resource and effort for testing in some degree. But does this mean that less resource, time, and effort will be needed for the testing activity in general or the cost for testing will be reduced. This issue will be discussed in chapter 5.

Chapter 4

# Process Related Test Automation Issues

## Introduction

This chapter will try to answer the question: *How can the MPC test process be improved in order to achieve efficiency & maintainability of test automation?* We want to know this because MPC are using automated testing to some degree but with limited success. Dustin et al relates the success of a test automation effort to the process of implementing the test tool [1] and we want to investigate if the testing process at MPC is lacking something that could help to improve the success of its automation efforts. The chapter begins with an overview of the development process used at MPC and then describes some of the related test documents in respect to automated testing. After the process description we will present our improvement suggestions that aim to make the test automation effort more successful.

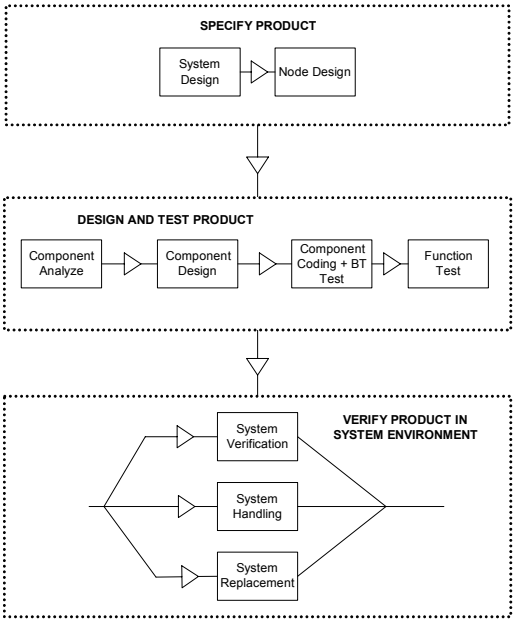
Our focus has been on general Time To Market (TTM) process documents and documents related to the current development of MPC's SMLC node: SMPC

## Process Description

The TTM process is common to all kind of developments at Ericsson whether it is technology, platforms or applications [11]. It supports activities from idea to a fully deployed product. The TTM is still in development but it is also used in the current development of MPS.

The parts of TTM that are of most interest to our work are similar to most modern development processes:

- Specify Product
- Design & Test Product
- Verify Product In System Environment




# Specify Product

## System Design

This phase starts with system design, which focus on requirements, architecture and risks. A requirement analysis is performed as well as an improvement analysis of the current products. This results in a requirement specification and a pre study report containing cost estimations, lead-time estimations, implementation proposals and so on. If the product will have impact on other nodes or projects, contact will be made with the appropriate teams.

There are no activities for analysing how to be able to perform automated testing on design, source or software binaries.

## Node Design

When system design is finished it is time for the node design phase. In this phase the designers will identify components needed to realise the system architecture and map requirements to each component. Implementation proposals will be written and interfaces between components will be specified and documented. The designers have to estimate the time to implement their design proposals and divide the implementation into drops.

Any new 3rd party products, test tools or simulators needed must be identified during the node design. There is no description of what actions are taken when these tools have been identified. Neither is there any information about existing test tools. Like system design there are no activities for analysing how to be able to perform automated testing on design, source or software binaries.

# Design & Test Product

## Component Analysis

This phase begins with an analysis of each component identified in the previous phase. The traffic flow of each component is analysed and use cases are made. A

functional description of the component is created and the interface input and outputs are identified. A test plan for the component is also written which we will look further into, later in this chapter.

### Component Design

In this step the designers will create detailed function and interface descriptions for all components that make up the node. Design documentation such as collaboration, sequence and class diagrams are created and included in the component implementation proposal. Events and statistics of the components are described as well as a detailed implementation time plan. Last but not least is a detailed test plan for each component created.

### Component Coding + BT test

This step will realise the component design into solid code, thoroughly tested in isolation. The components will be implemented and basic tested. A component description for each component must be written.

The details of basic component testing are described in chapter 3. Each component's detailed test plan created during component design explains how to basic test the components respectively.

### Functional Test

This step will test the whole node with all its components but not together with other nodes in the system. The complete node is supposed to be functionally tested by using the requirement specification as a base for test cases. User guides and installation manuals are also verified and approved in this step. The process cannot continue until all test cases has passed and all major trouble reports have been solved.

## Verify Product in System Environment

This phase has three parallel steps:

- System verification
- System handling
- System replacement

There exist very limited documentation about this phase but this is basically where system testing is performed. Deployment preparations are also started during this phase. The organisation needs to be prepared for taking orders for the product, training programs must be created and so on.

# Test Plans

The documents related to the TTM that are of most interest to us are the test plans. We have studied these in order to find out how they handle the issue of test automation.

## Main Test Plan

This document is basically the test strategy and gives descriptions of the type of tests that should be carried out; basic test, system design test, function test, system test on node level, system handling and system replacement. The basic strategy is to find and correct errors in an as early stage as possible, which means reduced cost in corrective measures [9].

Tools that should be used for each kind test are listed and each test uses at least one tool. “In Basic Test and System Design Test the designers are responsible for correcting the errors directly and re-execute the test cases until passed” [9]. Tool supported trouble reporting is not introduced until function test. The main test plan outlines all internal deliveries between process phases and states that test tools must be updated before each delivery. The main test plan does not deal with roles and responsibilities other than those of designers responsible for basic testing of components and team leaders responsible for system design testing. There is no mentioning of how to manage the basic test testware development other than scripts used to describe the test cases.

## Component Test Plan

Each component in SMPC has a test plan for its basic test. Even though the focus on this test plan is on the basic test it also mentions “important classes shall be unit tested by the designer” [18]. The aim is to automate the verification of test results as much as possible but the use of automation is not required. However, all methods in the important classes must be verified to work as expected. To define the important classes is left to the designer. How a unit test is carried out is left to the individual designer to decide. The designer shall write a unit test summary, which describes the unit testing that has been carried out. If any test automation techniques has been used these should be included in the summary as well as the achieved test coverage.

The test component developed for basic testing is described in detail as well as the test cases it supports and how to perform the basic test. The test results are supposed to be entered into the test plan along with the test cases.

# Improvements

As identified in the background chapter, Dustin et al [1] pointed out “over the last several years, test teams have largely implemented automated testing tools on projects without having a process or strategy in place describing in detail the steps involved in using the tool productively”. In the environment described by Dustin et al it is likely to believe that the success rate, when introducing an automated test tool, would be low. According to the research of Fewster et al [7] half of the organisations that have purchased a test execution tool have not achieved any benefit from their investment because their tools have ended up not being used. These kinds of software products are something Fewster et al [7] calls *shelfware*. Clearly there has to be a relationship between the way automated testing tools are introduced to organisations and the introduction success rate.

By studying the TTM and its related documents as well as literature and articles on the test automation topic we could identify issues where improvements to the TTM could be made. Early on we concluded that it was out of our scope to try to change the actual steps in TTM but rather to find missing parts or parts that needed improvement modifications. Not all of the improvements that we identified will be included. We were limited by scope, time and space and chose to explain only those that would neatly fit within these limitations.

This section will present some issues both related to the process and the MPC test automation objectives. We believe these issues have one thing in common: they can be used to improve the success rate of using test automation in an organisation. We have identified the need for the following improvements:

- Design for testability
- Define and communicate objectives
- Framework for unit testing

The first two improvements were selected because we believe they aim at improving the test automation in general, independent of techniques. They will contribute to the foundation of a development environment where test automation in general can be used more successfully and also help reaching the test automation goals set by MPC. It is more efficient to implement automated testing on a system that is prepared for it and has features supporting it than adding automated testing on top of a system that is not built in a way that will make it easy for testers to perform tests.

The third improvement was chosen because the process requires unit testing but only encourage this activity to be automated but with a framework it will be easier to actually achieve this goal.

## Design for testability

One of the objectives MPC has for its test automation effort is to increase efficiency of its test automation efforts, as explained in the background chapter. We believe that if test automation is taken into account during product design by taking steps to increase testability that will help test engineers build automated tests, the test automation efforts will be more efficient.

IEEE defines software testability as "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met, and the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met" [22]. We interpret this very formal definition as testability being a measurement of how easy it is to test a piece of software.

Pettichord [31] claims the two core aspects of testability as:

- **Visibility:** Our ability to observe the states and outputs of the software under test.
- **Controllability:** Our ability to provide inputs and reach states in the software under test

An example of how visibility and controllability affects testability is the use FDS in MPC products. The basic testing can be performed with an automated tool such as the DailyTest component because of the testability of FDS. FDS is designed in such a way that it is possible to replace real product components with automatic test tools. It is of course the most ideal situation when standard functionality of a product increases the testability in itself. Other kind of testability built into a product might not be directly needed for product users and thus add complexity which in itself can increase the risk of introducing even more defects. It is therefore necessary to weigh the benefits certain testability features give test engineers to the risk of introducing more defects. FDS and the DailyTest tool are covered in chapter 2.

Improving the installation of a system by for example allowing unattended script driven installation, can increase the system's testability. Testers can faster and easier set up the system for testing. Even if they are only doing manual testing this improvement allows them to automate tedious set up procedures. Another example of testability is to support configuration files in plain text. This makes it easier to set up before each test case and test in several different configurations.

A simple modification to the TTM process to increase the testability awareness during system, node and component design is needed implement this improvement. Testers need to state their requirements for what they need to make testing of the product easier and give feedback on designs that will be harder to test. Even though some of their requirements cannot be met, designers will learn more about testability, which will pay off in future projects.

We believe maintainability is a software quality attribute that not only affects the owner of a software product in terms of maintenance cost but also affects the developers in terms of ease of maintenance. We have the same opinion about testability because it affects the ease of testing which in turn affects the cost of testing. Testability is a software quality attribute that is just as important to the development organisation as maintainability

We also believe it is a good idea to increase testability by making sure all requirements are testable even though it is not strictly concerning test automation

## Framework for unit testing

As described in chapter 3, Smale identified the need for test case writing standards. As specified by the test plan Today MPC designers can choose by themselves how to write their unit tests [17], making it harder for new designers to carry on the work of others. Instead of letting MPC designers choose how to write their unit tests we suggest that a unit-testing framework is chosen and whenever unit testing is used it shall be based on this framework. Training is needed to make designers understand how to use the framework in order to make it easier for them to adapt to a standardised way of performing unit testing.

A unit-testing framework will make it easier to get better maintainability of unit test automation testware than with ad hoc unit testing that is different for each designer. Ad hoc unit testing is often accomplished with code assertions and debugging output placed inside the unit of code being tested. These methods are hard to maintain because they are test case independent. A unit-testing framework will allow the designer to separate test code from product code that will give a cleaner and more maintainable product code. Using unit-testing frameworks, designers can write several test cases for the same piece of product code, testing it in different ways, allowing for more flexible and maintainable testware.

Another improvement that can be achieved by using a framework is that it will be easier to calculate test coverage since this feature is often included in the framework, or can be found in a support tool specifically built for this. The designers will benefit from this since they are supposed to include the test coverage in their test reports when using test automation to test their components, as described in the component test plans [18].

A framework will also make it possible to include this type of testing in the daily build process MPC aims to use.

The improvement we suggest here involve the following steps:

- Selection of unit testing framework

- Training designers in using the framework
- Explicitly specify usage of framework for unit test development

## Define and Communicate Objectives

Test automation objectives define the goals for test automation efforts. During our background analysis we found that efficiency and maintainability was important objectives for test automation at MPC. If these goals are to be achieved they have to be communicated so that all the people, be it managers or designers, involved in test automation can work towards achieving these goals. Pettichord points out "goals will often differ between development management, test management, the testers themselves and whoever is automating the tests. Clearly success will be elusive unless they all come to some agreement." [19]. If the objectives for test automation is not clearly stated the parties involved are likely to have different, often contradicting, expectations. It is likely to believe that managers will have expectations with focus on resource and organisational goals and designers will have expectations related to more individual goals. A more diversified view of possible expectations among managers and designers than those listed in chapter 3 are suggested by Pettichord [19]:

- Speed up testing to accelerate releases
- Allow testing to happen more frequently
- Reduce costs of testing by reducing manual labour
- Improve test coverage
- Ensure consistency
- Improve the reliability of testing
- Allow testing to be done by staff with less skill
- Define the testing process and reduce dependence on the few who know it
- Make testing more interesting
- Develop programming skills (among testers)

With or without clearly defined objectives it is possible that the objectives are unrealistic or wrong. When actually defining the objectives and incorporating them into the test process it necessary to validate that the objectives are realistic and correct, something that individuals probably will not do if there are no official objectives and they have to make up their own.

We believe that by defining objectives, such as those described in chapter 3, for test automation it will be easier to achieve success since there is a clear goal for everyone in involved to aim for. Without defined measurable objectives it is not possible to know when an objective is met and it is possible to set new objectives to achieve.

# Conclusions

In general there are few steps in the TTM that aims to support the use of automated testing. We have suggested a few improvements but more can always be done.

To succeed with test automation and benefit the most from it, it is important that as many test cases as possible are possible to automate. By designing testability into

the product and making it easier to test can help making the test automation effort a success. The less effort needed to implement and maintain an automated test the greater the benefit from the automation will be in terms of efficiency and maintainability. The use of the DailyTest tool and FDS is a good example of the benefits that comes with testability.

Certain steps need to be taken to increase the testability and leverage its benefits. The process must be altered in such a way that the test team is involved in the design phase. This has to be done to assure that the test team will be able to influence the outcome in such a way that it has support for automated testing. There is always a risk of making the design more complex by adding testability features. Whether this is acceptable or not has to be decided on a case-to-case basis.

It is out of scope to go into the specific design changes so we have left this for further research.

Another key issue regarding the success of test automation is management of the objectives. To be able to reach the objectives set for the test automation effort they have to be communicated to all team members involved in test automation. Without common goal to work towards it is unlikely that any goal will be achieved.

Chapter 5

# Test automation and cost reduction

## Introduction

This chapter will answer our research question: *Is it possible to reduce cost of testing with test automation?*

MPC suspects that it is possible to save money with test automation but they are not sure. They want to know other organisations' experience of this subject and hopefully get more proof that their suspicion is correct. A deeper analysis of the economical effects of automated testing would probably be more correct but that would involve economists and mean more work than available to us. Therefore we will present the opinions about this subject found in the test automation community as well as our own thoughts on this subject. The focus is not on how to apply these opinions to MPC test process but rather show different viewpoints on this chapter's research question.

## Cost of testing

Why reduce the cost of testing? Performing software testing is a process that cost money and resources and generally does not generate any income. It is not so questionable why any organisation would like save some of this money and resources in order to make a better profit in the end. Hoffman [3] identified the following activities that make up the cost of doing manual software testing:

- Staff training
- Product analysis and learning
- Test case design
- Test executions
- Test result analysis
- Test management

How is it possible to reduce the cost of these activities? One way of reducing these costs is to just skip some of them. Skipping any of the test creation, execution or analysis activities would make all of the other activities worth zero. Skipping the inspection, training or management part could probably be done but things could get tricky because people would not know what to do or how to do it. What about doing less? That would certainly work but quality would suffer from it. We obviously do not think any of these solutions are good enough, but we do think that to reduce the cost of manual testing, it has to be performed more efficiently. What about costs for automated testing? Some costs for automated testing that were identified are:

- Test case design for automation [3]

- Test case implementation [3]
- Automation environment design & implementation [3]
- Maintenance of testware & environment [3]
- Automated tool licenses & support [3]
- Automated tool introduction & training [3]
- Automated tool operation [3]
- Oracle creation [3]
- Test process analysis and change [1]
- New tasks necessitated by test automation [25]
- Identify test automation candidates

Some of the test automation costs relate to the introduction of test automation and others to the continuing use and maintenance of test automation. The costs related to the introduction process are more or less fixed while the costs related to usage and maintenance is variable over time. There is however no consensus about whether these costs can be measured in money figures. Bach [25] argues that "hand testing and automated testing are really two different processes, rather than two different ways to execute the same process" [25], which he means would make it meaningless to do direct comparison between the two of them. Hoffman [26] on the other hand, claims it is possible to quantify the costs of test automation and manual testing and provides formulas for calculating return of investment. Hendrickson's [23] experiences tell her that it is difficult to measure the benefits of test automation in money as well as to put a price on test automation efforts.

## False expectations

Our own experiences tell us that when you mention test automation, people think of it is a mean to save money. It is likely to believe that even asking the question, if test automation can reduce cost of testing, stems from a wish that it could do that. Hayes [27] claims the reason for test automation is not to "reduce either test resources or cycle time" but rather "reduce the risk and cost of software failure by increasing test coverage" [27] hinting that the benefits lies beyond the test phase.

Fewster et al suggests that regression tests should be automated which would "more quickly yield an automated Test Suite with greater potential for payback" [7]. The reasoning behind this is of course that these tests can be executed faster and with less effort from test engineers. We realise that in some cases this will be cheaper than running the same tests manually. But as shown in the previous section there are more costs to test automation than solely test execution and operation of test tools and they must all be taken in account. The regression tests needs to be designed, implemented and maintained. One important benefit from being able to run automated regression tests often is that you will more quickly find out when defects are introduced in the software. The faster you fix a defect the less impact will it have on the software. This is valuable because the sooner a defected can be identified and removed the greater is the saved cost of fixing the defect which has existed for a long period of time. It is the same reasoning that fixing a defect in the requirement specification is cheaper than fixing a defect found after delivery. This is prevention of cost increase, not cost reduction.

Dustin et al [1] claims that the introduction of test automation will not lead to immediate test effort reduction but rather increased effort to begin with. This

increase in effort originates from all the new activities that test automation adds to the test process, as described in the previous section.

Bach [25] and Hayes [27] claim that they have never seen staff or schedule reduction as a result of using automated testing tools. Our studies point in another direction as for what the real benefits of test automation is.

# Test automation benefits

What are the benefits that could be achieved with test automation? Some indirect answers to this question were included in chapter 3 where a list of tests that could benefit from automation was presented. Before answering this question, one should ask oneself, why automate in the first place? Linda Hayes describes in the book Software Test Automation [7] three key benefits that automated testing provides; repeatability, leverage, and accumulation.

The term repeatability refers to tests that can be executed more than once, consistently each time.

The term leverage refers where else to get true leverage of test automation such as automating tests that were never performed manually at all because it was not possible to do it manually.

The term accumulation relates to how important and critical it is to adopt an approach to test library design that supports maintainability over the live of the application.

## Discussion

The benefits of automating tests could be summarised;

- Tests can be run faster
- They are consistent
- Tests can be run over and over again with less overhead.
- They allow performing tests that are impossible to do it manually.
- They improve staff moral and confidence
- Testing elapsed time can be shortened
- Frees skilled testers to put more effort into fix bugs and design better test cases to be run.
- Production of a reliable system.
- Improvement of the quality of the test effort.

As more automated tests are added to the test suite more tests can be run each time thereafter. Manual testing will never goes away, but these efforts can now be focused on more rigorous tests.

Benefits from automating different types of tests described in both chapter 3 and above are obvious. But the question is how often you will execute these kinds of tests over the life of the application or even within the same release cycle. If you manage to find out that there exist such tests that could be performed repeatable over long period, does it mean that automating these kind of tests will led reducing testing time, resource and effort for the entire test cycle time.

Automation is not cheap and requires investment in terms of both time and money whether you select and buy a tool or develop one of your own and maintain

throughout the projects. Zallar [30] points out “the effort of test automation is an investment”. More time and resources are needed up front in order to obtain the benefits later on. Zallar [30] says also that in some occasions, is it possible to get a immediate payback, but these opportunities are usually small in number relative to the effort of automating most test cases.

But as Fewster et al [7] mentioned before, once a set of test has been automated, it can be repeated far more quickly than it would be manually, so the testing elapsed time can be shortened (subject to other factors such as availability of developers to fix defects). This in turn may lead to earlier time to market.

Linda Hayes [27] claims that the real payoff to justify test automation can be found by looking at:

- The cost of failure for the system you are testing
- What does it cost the company or its customer in time, resources, and money if defects escape to production?
- And what is worth to deliver on time with quality product.

Dustin et al [1] claims that the main purpose for automating tests is to do more with less in order to increase the profitability of the software product by cutting both resources necessary for testing and also due to a shortened time to market. How could this be realised with test automation?

If cutting resources necessary for testing means:

- To free many testers from doing repetitive, time consuming tests
- To use regression tests that could test the basic functionality faster than manually testing
- To use test automation for tasks that require more resources and time to perform than what’s available, such as stress and capacity testing your system.

Then it’s obvious that test automation will help in these particular situations.

If shortened time to market means that the use of automated testing tools will free skilled testers to put more effort into fix bugs which in turn may lead to shortened time to market, this could be true. But, is this really a common think to happen in all the companies that uses automated testing tools? This could also be hard to quantify.

Benefits with automation do effect the testing time, resources, and effort but not in such a manner that less testing time and resources will be needed for the entire test cycle. The time saved with test automation could be used;

- To fix more bugs.
- To test more often.
- To reduce the risk and cost of software failure by increasing the test coverage.
- To put more effort into design, build, and maintenance of test scripts that will be run.
- To train other testers so that they can use and benefit from the tool.
- To inform managers and other stakeholders about the test progress.
- To carefully plane and manage the testing activities.
- To improve the test automation strategy.

We believe that test automation effort must be made with a strategy and in the long run, as any immediate payback is eaten up by the initial costs. Focus should not be on reducing the test budget but rather prevent unexpected quality costs.

# Conclusion

The costs of manual and automated testing can be quantified respectively but a comparison is harder to do because they are not interchangeable. In the same sense it is difficult to quantify the benefits of automated testing compared to manual testing. Quantification of automated testing benefits very hard to do since there are no clear definitions in economical term.

Test automation can help testers in their pursuit of finding defects in software and we believe this is the most important aspect of test automation, not as a mean to save money.

We believe that test automation, if used correctly, can be an instrument to either get greater test coverage or shorten the amount of actual testing. If test automation is developed in parallel with the product it is supposed to test we believe this will give test managers the ability to efficiently test certain parts of the products which will save them time and effort. This benefit can be used to either shorten the testing time or be used to test parts of the product not yet covered by testing. This does not mean that test automation will reduce costs directly but will help to shorten the time to market.

Chapter 6

# Evaluation of automated testing techniques

## Introduction

The main purpose of this chapter is to find out what automated testing techniques to use in order to achieve efficient and maintainable test automation. We will evaluate different automated testing techniques such as scripting and comparisons in order to be able to decide which techniques are most appropriate for MPC. Techniques that will make it possible for MPC to achieve their test automation objectives, thus making test automation more efficient and maintainable.

Other techniques such as, capture replay is also a technique that is used to automate test. The test tool captures the actions and records them in a script, which can then be replayed to drive the application automatically. But this technique is not a good basis for long-term test automation. Groder describes in Software Test Automation [7] projects that use capture/replay as a design methodology, which result that the projects had been failed due to maintainability issues with the test code. Since maintainability is important issue for MPC, capture replay is not appropriate for MPC.

## Automated testing techniques

### Scripting techniques

According to Fewster et al [7] script is a form of computer program, a set of instructions for the test tool to act upon and the main purpose for scripting technique is to increase productivity, and make test automation easier. But yet there are some difficulties to overcome in order to make sure that the scripts fit for their purpose. One of the major problems with scripts is how to deal with the maintainability of test scripts. If the scripts are going to be reused by lots of different tests that are going to have a long life, then it will be worth ensuring that the script works well and is easy to maintain. This is what MPC want to achieve by making their test scripts more maintainable and related to MPC's second objective for test automation that is, to increase the maintainability of test artefacts that is used to automate tests.

Principles for good scripts identified by Fewster et al [7] are:

- Annotated, to guide both the user and the maintainer.
- Functional, performing a single task, encouraging reuses.
- Structured, for ease of reading, understanding, and maintenance.
- Understandable, for ease of maintenance
- Documented, to aid reuse and maintenance.

Bergdahl [8] describe properties that scripts used by the Daily-Test tool should have:

- Easy to write
- Easy to read by humans
- Support for extensive commenting.
- Usable for testing

As you can see, all the steps above by Fewster et al [7] and Daniel Bergdahl [8] are one way or another relates to the maintainability of test scripts. The reason why the maintainability issue is so important here is to avoid a higher maintenance cost than the equivalent manual testing effort.

According to Fewster et al [7], one of the benefits of editing and coding scripts is to reduce the amount of scripting necessary to automate a set of test cases and this is achieved principally in two ways; The first one is coding relatively small pieces of scripts that each performs a specific action or task that is common to several test cases. Each test case that needs to perform one of the common actions can then use the same script. The second one is inserting control structures into the scripts to make the tool repeat sequences of instructions without having to code multiple copies of the instructions.

There are different scripting techniques that seek to achieve a way to improve the maintainability of scripts, reduce the size and number of scripts and their complexity. We will go through some of the techniques, their pros and cons presented by Fewster et al [7].

## Linear scripts

Linear script is what you end up with when you record a test case performed manually.

It contains all the keystrokes, including function keys, arrow keys, and the like, that control the software under test, and the alphanumeric keys that make up the input data.

A linear script may also include comparison instructions while the test case is being recorded (if the tool allows this) or it may be done in a separate step afterwards, perhaps while replying the recorded inputs from the script.

The technique can be used for any repetitive action, if test case will only be used once, for demonstration or training, and to update automating tests.

### Advantages of linear scripts

No up-front work or planning is required. You can just sit down and record any manual task and quickly start automating. It provides an audit trail of what was actually done. The user doesn't need to be a programmer (providing no changes are required to the recorded scripts, the script itself need not to be seen by the user). The technique is good for demonstration (of the software or the tool)

### Disadvantages of linear scripts

The process is labour-intensive. It can take 2 to 10 times longer to produce a working automated test (including comparisons) than running the test manually. Everything tends to be done from scratch each time. The test input and comparisons are hard-wired into the script. There is no sharing or reuse of scripts.

Linear scripts are vulnerable to software changes, and are expensive to change (they have a high maintenance cost).

## Structured scripting

There are three basic control structures supported by probably all test tools scripting language.

The first is called ‘sequence’ and is exactly equivalent to the linear scripting approach in the sense where the first instruction is performed first, then the second, and so on.

The second one is the selection script that gives ability to make a decision whether to execute one statement or the other.

The third one is iteration that gives a script the ability to repeat a sequence of one or more instructions as many times as it required. Sometimes referred to as a ‘loop’ there the sequence of instructions can be repeated a specific number of times or until some condition has been met.

In addition to the control structures, one script can call another sub-script, which executes and then returns to the point in the first script immediately after where the sub-script was called. This mechanism can be used to divide large scripts into smaller, and hopefully more manageable scripts.

### Advantages of structured scripts

It will increase the amount of reuse and even make scripts more flexible. It leads to maintainable and adaptable scripts that will in turn support an effective and efficient automated testing regime. Scripts can be made more robust and can check specific things, which would otherwise cause the test to fail. Structured scripts can also deal with a number of similar things that need to be repeated in the script using a loop. And finally structured scripts can also be made modular by calling other scripts.

### Disadvantages of structured scripts

Script has now become a more complex program and test data is still hard wired into the script. Programming skills may be required in some degree.

## Shared scripts

As the name implies, they are scripts that are used (or shared) by more then one test. The idea is to produce one script that performs some task that has to be repeated for different tests and then whenever that task has to be done we simply call this script at the appropriate point in each test case.

There are two types of shared scripts;

- Those that can be shared between tests of different software applications or systems. An application-independent scripts may be more useful long term, and worth putting additional effort into.
- Those that can only be shared between tests of one software application or system.

The use of shared scripts is one step towards building automated tests that can keep up with rapidly changing software, i.e. that will not require extensive maintenance.

### Advantages of shared scripts

Similar tests will take less effort to implement. Maintenance costs are lower then linear scripts. Eliminates obvious repetitions and can afford to put more intelligence into the shared scripts.

### Disadvantages of shared scripts

There are more scripts to keep track of, document, name, and store. If not well managed it may be hard to find an appropriate script. Test-specific scripts are still required for every test so the maintenance costs will still be high. And finally shared scripts are often specific to one part of the software under test.

## Data-driven scripts

Stores test inputs and the expected outcomes in separate (data) file rather then in the script it self and when the test is executed the test input is read from the file rather than being taking directly from script. But this leaves the instructions, control information in the scripts. A significant advantage of this approach is that the same script can be used to run different tests that use same instruction, control information but different input and different expected outcomes.

### Advantages of data-driven scripts

It is possible to implement many more test cases with very little extra effort. The format of the data file can be tailored to suit the testers. Data file can be allowed to contain comments that the scripts will ignore but that will make the data file much more understandable and, therefore maintainable. The inputs and the expected results are removed from the script and put into data files where expected results can be directly associated with a particular test input. Testers without technical or programming knowledge about the tool scripting language can do adding new tests. And finally there is no additional script maintenance effort for the second and subsequent tests.

### Disadvantages of data-driven scripts

Writing the control scripts does need to be done by someone with a technical (programming) background. If your testers are not comfortable with programming automated test scripts then the skill has to be brought into the team. The initial set-up will take some time. It must be well managed. Finally if you don’t have many tests, this approach will entail more work, and will seem excessive. This approach is not appropriate for small systems.

## Keyword-driven scripts

The technique is a more sophisticated data-driven technique. It allows taking out some of the intelligence from the script and putting it into the data file. This would permit a single control script to support a wider variation in what the associated test cases could do but the complexity of test files will increase vastly. With this approach, we do have test files rather than data file since test files describe the test case. The control scripts read each keyword in the test file, and then calls the associated supporting script. The supporting scripts may require more information from the test file, and this can be either read directly or passed on by the controlling script. The control script is no longer tied to a particular feature of the software under test, nor indeed to a particular application or system.

The main difference between Keyword-driven scripts and those described earlier is that there are two fundamental approaches to test case implementation, prescriptive and descriptive:

- Key-driven scripts allows us to take a descriptive approach to implement an automated test case there we have only to provide a description of the test case more or less as we would do for a knowledgeable human testers. This approach states what the test case does in the test data but not how it does it. Then the control scripts reads what the test case does and calls the associated supporting script.
- Others described earlier used prescriptive approach to implementation of test cases, there how the script will do the task (the intelligence) is included in the scripts it self.

### Advantages of keyword-driven scripts

The advantages of this approach are extremely large and they are for example, many more tests can be implemented without increasing the number of scripts once the basic application supporting scripts are in place. This greatly reduces the script maintenance cost and not only speeds up the implementation of automated tests but also makes it possible to use non-programming testers to implement them. Thousands of test cases will be implemented with only a few hundred scripts. The technique can be implemented a way that is tool independent (and platform independent). If the software under test has to be tested on different hardware platforms that are not all supported by the same test tool we will not have to change our tests merely the supporting scripts. Finally the way in which test are implemented can be tailored to suit the testers rather than the test tool by using the format and tools that the testers are most comfortable with. It is then possible to optimise the format so that new automated tests can be implemented as quickly as possible and yet in the safety way.

### Disadvantages of keyword-driven scripts

The complexity of the data file would increase vastly and if not managed well, it would most likely outweigh any benefits. If the tool does not support reading data from other source, then you will need to develop your own approach. You may want to develop your own approach even if your tool supports it since you will then be independent of any tool.

## Scripting Techniques that are most suited for MPC

What MPC want to achieve with scripts is to make test automation easier, be able to reuse test scripts between different applications/versions whenever possible and of course to make their test scripts more maintainable in order to avoid a higher maintenance cost than the equivalent manual testing effort. Therefore, different scripting techniques described above offer one way or another to achieve maintainability of test scripts and make test automation easier. In order to be able to decide which of the technique is most suited for MPC, some criterion will be developed that we will base on our decisions towards techniques that are appropriate for MPC.

These are the criteria used for selection:

- Scripting technique that allows a way to reduce the amount of scripting necessary to automate a set of test cases.

- A technique that support coding relatively small pieces of scripts that each perform a specific action or task that is common to several test cases and could be reused whenever possible.
- A technique that support a way to insert control structures into the scripts or test data to make the tool repeat sequences of instructions without having to code multiple copies of the instructions.
- A technique that provide a way to reduce the size and number of scripts and their complexity.
- A technique that support a way to structure scripts for ease reading, understanding, and maintenance.
- A technique that support for extensive commenting.

The reason why we choose this criteria above to single out appropriate techniques for MPC is that the criteria includes important characteristics that should be part of techniques that will be able to facilitate a way to achieve maintainability of artefacts used to automate tests. Therefore, we believe that the use of these criteria will be sufficient for the current situation than adapting other criteria out there that may not include be suited for this particular situation.

Scripting techniques described above were Linear, Structured, Shared, Data-driven, and Keyword-driven. Linear scripts are not appropriate for MPC, because the process is labour-intensive, it is vulnerable to software changes and they have a high maintenance cost, and they do not support sharing or reuse. But the other four remaining provides a way to achieve maintainability, or to reduce the size and number of scripts and their complexity. Before we go further to decide which technique that is appropriate for MPC, let us have a look how scripts are structured within MPC and which scripting technique that is in use.

MPC department create their own automated testing tool (Daily-Test tool) instead of purchasing a ready automated testing tool, and therefore a script language where developed for the tool in order to simplify the input of test cases for the tool. According to Bergdahl [8] the term script is referred to as a pure command sequence than a shell script and advances scripting features such as conditionals, loops are missing but could be developed for the next version for the current script language that is in use.

Bergdahl [8] states that script where used as input of test in order to make the input of test cases as easily as possible. The Daily-Test tool reads in the test scripts, interpret the information found in these scripts and then calls the associated supporting code.

Bergdahl [8] describe how scripts used within MPC there scripts where structured in a way that is a line based. One line equals one test there each line is separated into three parts:

- The test specification that describes what test to perform.
- Test action that describes what action that should be performed
- And variable (s), arguments for the test that contains path for the input, expected outcomes and test files.

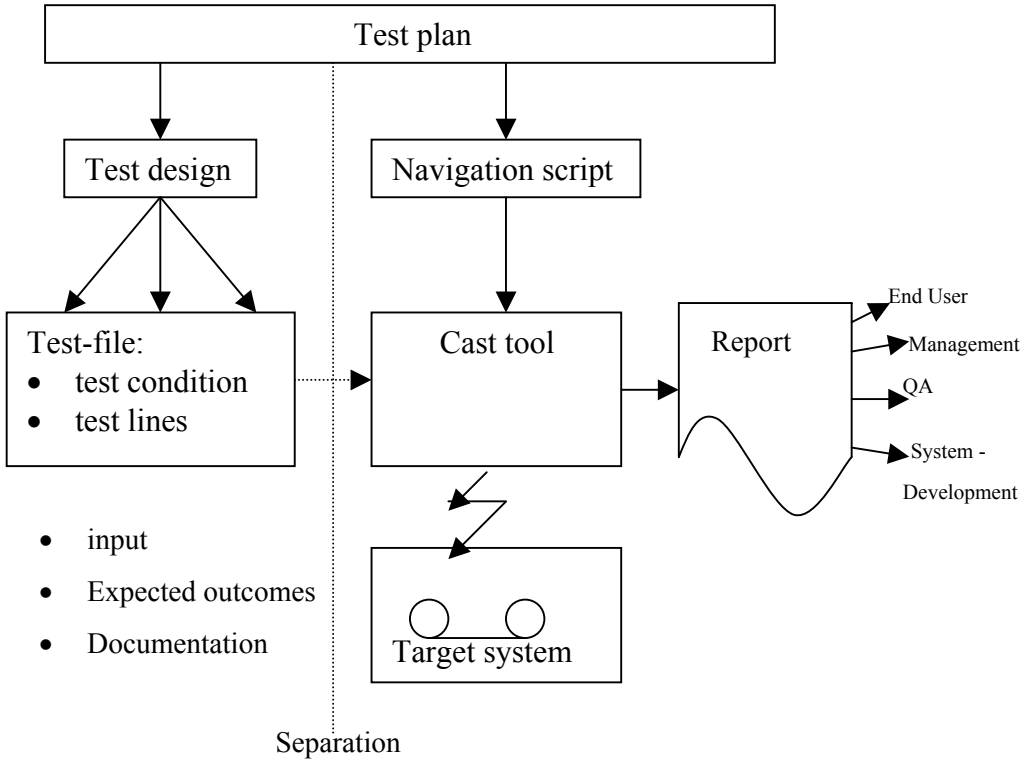
In addition, comments were allowed everywhere in the same way as you can do in C/C++.

According to Bergdahl [8] the main function of the tool is to run tests towards other components by executing the instructions within these scripts in order to test different functionalities of the component over and over again. The creation of

inputs, expected outcomes and test files will be prepared separated from the tool by designers/testers for each component that will be tested.

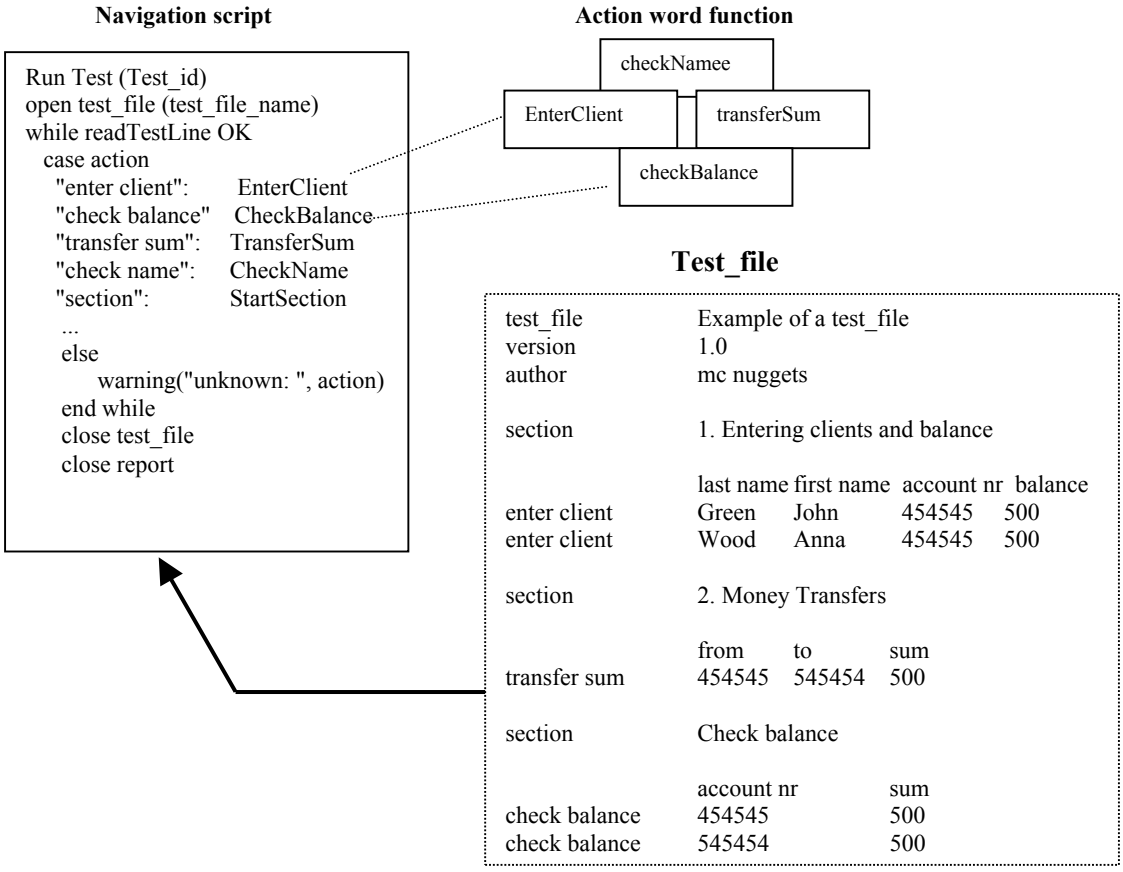
The script technique that is used within MPC is much like the data-driven scripting technique (See the last point in the list above where test input and expected outcomes stores in a separate data files instead of being part of the script) and even includes features that indicate that the technique is also towards Keyword-driven technique (See the second point in the list above where an action word/keyword is used to decide which action to perform). The basic principle for the technique used within MPC and Keyword-driven technique is the same where both techniques keeping design of the test input, expected outcomes and test files separated from the automation of the test. The focus is not on the test tool itself or the scripts to describe tests but the tool is used intensively to execute the tests, the area where test tools are usually very good. The tests themselves are described outside the tool, in so-called test file.

Buwalda presented a schematic overview of the basic principle of the Keyword driven scripting technique in the book Software Test Automation [7].




But the way keyword-driven technique is implemented within MPC differ from how other authors such Fewster et al [7] describe the implementation for the technique.

Fewster et al [7] claims that a special script (navigation script) is written to interpret and execute the information in the test-file that describe what is going to be tested and then calls the associated supporting functions. The following figure by Fewster [7] shows the relation between navigation scripts and data files




Scripts used within MPC does not function as a navigation script but the tool itself (Daily-Test tool) reads in the scripts, interpret the information found in these scripts and then calls the associated supporting code. According to Bergdahl [8] the scripts used within MPC contains information such as what component to test, test actions, variables that holds the path to the test files, and instructions for suspending test (DELAY), configuring the test, loading files, create, start, stop, and delete component.

The reason why the implementations of the technique differ might depend on the kind of applications that uses the technique. Different organisation might develop different kinds of application and implement the technique in different ways while the basic principle of the technique is still in the same, thus keeping design of the test strictly separated from the automation of the test.

We believe that the keyword-driven technique is appropriate for MPC and is the right way to go for several reasons. The technique allows keeping test separated from their execution there a collection of tests that have more or less the same scope and level of detail are identified, designed, and created. Features that could be performed with structured, shared, and data-driven scripts are also possible with Keyword-driven. Tests are independent from the tool and scripts and the approach allows creating maintainable and structured test set and automating its execution in a reusable way. More tests can be implemented without increasing the number of scripts, once the basic application supporting scripts are in place. Tests are easy to read because all details needed for their execution are hidden behind the action words and actions executed by the navigation scripts/Daily-Test tool could

sometimes be the same, but with different data/input and expected result for every test line. Finally, to adapt the technique in the MPC organization will not be a difficult issue because similar technique or a technique towards keyword-driven where already in use within MPC.

Buwalda and Kasdorp [28] describe how testing tasks should be divided into test files (test clusters):

| **Dividing Testing Tasks into Test Files (Test Cluster)** | | |
|---|---|---|
| Priority | Criterion | Explanation |
| 1 | Logic | The division should be perceived as logical by the people involved in making, reviewing, and/or maintaining the test |
| 2 | Independence | Execution of test files should generally be independent of the execution of other test files. When there are dependencies between test files, these should be the result of a well-considered decision |
| 3 | Type of test | The division should take into consideration the type of test to be done, e.g., module tests, system tests, functional tests, or performance tests. |
| 4 | Scope | The division should take into account the scope that has been decided upon in the test strategy. |
| 5 | Intended method of execution | Separate test files can be identified by the way test is probably is going to be executed (manually, automate with a record-and-play test tool, or automated with a C program) |
| 6 | Project issues | What functionality does the customer want to have tested first? When is necessary design information going to be available? In which order will parts of the system be completed? |
| 7 | Cluster size | To some extent, the size of the test clusters should be taken into consideration as well: If a test file becomes very large, consider splitting the file further. On the other hand, if the test file is very small, you might consider combining them |

Furthermore, Buwalda [7] identified risks that one should be aware of when using the keyword-driven technique in Software Test Automation:

- Incomplete or incorrect use of the method, especially when important basic principles like the separation of data and navigation are not addressed properly the results can easily be disappointing.
- Underestimation of the effort needed to interpret the test result. For example, the actual behaviour of a system can be different than expected in the test and it takes time to find out if this is a mistake it the system or in the test specification.
- Improper organisation of the test maintenance. Maintaining the test might not be much work because of the separation between test clusters and navigation, but someone has to do it, and needs the files to do it with and the knowledge of how to do it.
- Political problems. For example, when a project is not meeting its planned deadline, problems between the development team and the test team can easily occur.

# Comparison technique

## About comparison techniques

According to Fewster et al [7], comparison techniques are used to verify if the software product produce the correct outcome. This is achieved by performing one or more comparisons between an actual outcome of the test and the expected outcome of that test. With the simplest comparators you can compare only standard text files but with more sophisticated comparators you can compare more complex and specialised forms of data. Comparison basically tells you whether or not the two sets of compared data are the same. The term 'test passed' is used when the comparisons not find unexpected differences and 'test failed' is used when it does.

Fewster et al [7] states that verifying repetitive and detailed tasks is very boring and it is easy to make a mistake if performed manually therefore these tasks will be ideal for using automated comparison. On the other hand if a defect exist in an expected result, automated comparison will hide, not highlight, the same defect in the actual outcome.

Another important issue related to automated comparison is the test analysis burden. Test analysis that will be performed manually after the test execution. If the test results are not organised in such a way that is easy to go through, and find the bugs founded by the comparator, the failure analysis time can be considerable long which in turn can be a time consuming activity by itself.

Several important decisions to be made when planning automated comparisons that have a significant impact on both effectiveness and efficiency of your automated testing are identified by Fewster et al [7]:

- The information you decide to compare and how much of it you compare will affects the test cases' ability to detect defects, the implementation costs, and the maintenance cost.
- If we compare too much information the cost of implementing and maintaining each automated test case may overwhelm the benefit it can bring.
- If we compare inappropriate or insufficient outcomes then the value of the test case will be reduced, if not lost altogether.
- Good balance between sensitive tests (sensitive test compares more and more often) and robust tests (robust test compares more selectively) is also needed and affects the ability to detect defect, the implementation and maintenance costs.
- The more sensitive a test case the more likely it will need some updating when software under test changes.
- If we run a set of sensitive test cases there is a good chance that a number of them will fail for the same reason.
- The more robust our test cases, the more chances there is of a defect escaping our attention.
- The more comparisons there are to perform, the more time it will take to prescribe them all correctly.

Automating comparison is not as simple as just comparing the actual and expected data but many decisions such as those described above should be considered when designing and implementing it. Therefore, the following sections will go through different comparison techniques, what decision to make towards automated

comparisons, and ideas, suggestions related to how to improve the test analysis burden.

## Dynamic Comparison

According to Fewster et al [7], dynamic comparison is a comparison that is performed during the execution of test cases and is the most popular comparison technique from the point of view where it is much better supported by commercial test execution tools.

### Advantages

It can be used to help program some intelligence into a test case for example if unexpected output occurs, the test case will be aborted rather than allowing it to continue.

### Disadvantages

Dynamic comparisons take more time to create, are more difficult to write correctly and will require higher maintenance cost. Furthermore, embedding further commands or instructions into the test scripts will make the script more complex.

## Post-execution comparison

According to Fewster et al [7], post-execution comparison is comparison that is performed after test execution was completed. Test execution tools generally do not support post-execution comparison as well as dynamic comparison and often seems to be more work to automate the post-execution comparison. Post-execution comparison can be done in two ways;

- Active approach: When we intentionally save particular result that we are interested in during a test case execution for the purpose of comparing them afterwards.
- Passive approach: When we simply look at whatever happens to be available after the test case has been executed.

### Advantages (Active post-execution comparison)

The actual outputs are saved and could be important doing so specially when complex comparison is performed where the actual and expected outcomes may differ in expected ways. Comparison can be performed offline and on different machine from that used to execute the test cases. It permits a much wider range of tools and techniques for performing more complex comparisons. Other additional outcomes that are used only in the event of test failure as an aid to analysing the cause of the failure.

### Disadvantages

Test execution tool cannot be instructed to act differently depending on the results because the outputs will be verified at the end of a test case run. To make test execution tool perform the post-execution comparisons, we will have to specifically add the necessary instructions to the end of the script and this can amount to a significant amount of work particularly if there are a good number of separate comparisons to be performed. In practice the post-execution comparison tool is not very well integrated with the execution tool, if not all.

## Simple comparison

According to Fewster et al [7], simple comparison looks for an identical match between the actual and expected outcomes and any differences will be found.

### Advantages

It is simple, easy to specify them correctly so few mistakes are likely to be made. It is easy for other people to understand exactly what is, and what is not being compared. Usually much less work is involved in implementing and maintaining simple comparison.

### Disadvantages

If tests have outcomes that can be legitimately differ each time test case is run, simple comparison will cause these test cases to fail even though the differences can be safely ignored

## Complex comparison

According to Fewster et al [7], complex comparison enables us to compare actual and expected outcomes with known differences that we expect to see or that are not important to us and highlight other unexpected differences between them. Expected differences can be for example dates and times that is usually expected to be different and will be ignored.

## Sensitive versus robust comparison

Sensitive comparison allows you to compare as much as possible as often as possible but takes more time and effort to implement and maintain, needs more disk spaces, cause analysis takes more time, and is preferable used at higher level.

Robust comparison is about comparing only a minimum of information and preferable used al lower, more detailed level.

According to Fewster et al [7], making the right choices is important since if we compare inappropriate or insufficient outcomes then the value of the test case will be reduced. If we compare too much information the cost of implementing and maintaining each automated test case may overwhelm the benefits it can bring. But the question is which one is best? The authors' [7] emphasises that there is no absolutely correct answer to this question. This is one of the design decisions that one would make when designing an automated test. However, they recommend two possible strategies;

### First possible strategy

- Use Sensitive comparison mainly at high level where test cases should be sensitive so that they will pick up any changes, no matter how small.
- Use robust comparison should then be used mainly at the more detailed levels where each test focuses on some specific aspects of the software under test.

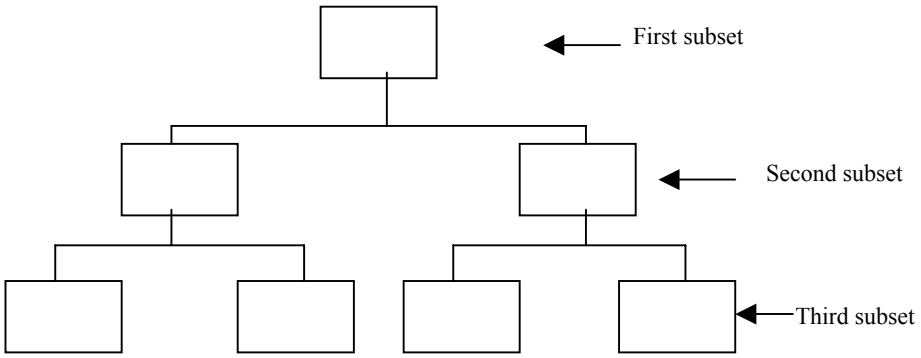
### Second possible strategy

This strategy is about to consider designing sets of test cases such that one or two of them use sensitive comparison while others use robust comparison. If one or two test cases use sensitive comparison then there is little point in the other test cases

within that set doing so again since one unexpected different in the sensitive comparison may cause all test cases within the set to fail.

## The test analysis burden

Fewster et al [7] state that in manual testing, the test analysis is done in parallel with the test execution but in automated testing, all of the analysis activity now becomes post-execution, and the more test to execute, the more test failures to analyse. It may be that as you analyse the tests, you discover that the test you are looking at now has actually failed for exactly the same reason that another test failed. In some cases you may in fact find that a large number of failures were caused by the same defect. The danger is that the analysis time remains proportional to execution time. A logical hierarchy to minimise analysis time was recommended by Fewster et al [7]:




The first level subset would contain the most general tests or breadth tests. The lower levels of subsets would contain the more investigative tests, the depth tests, tests that are probing more for errors. If any tests in the first subset failed then don't analyze any failures at the lower levels until all or most of the failures at the top level are fixed or at least thoroughly understood.

## Criteria for effective and efficient comparison

Guidelines for comparison that was identified by Fewster et al [7] can be used as criteria for implementing effective and efficient comparison;

- Keep it simple: Comparisons that are as simple as possible are less likely to either generate false failures or to miss true differences.
- Document the comparisons: It is important to insure that everyone using a comparison process understands exactly what it does and does not do.
- Standardise as much as possible: The more things are standardised, the easier it will be to automate comparisons.
- Divide and conquer: Small comparisons that are limited in scope are easier to set up and less likely to go wrong.
- Keep efficiency in mind: Use simple comparisons, which will complete faster than complex comparisons wherever possible. If the simple comparison fails it will then be necessary to perform the complex comparison.
- Avoid bitmap comparisons: A bitmap is also dependent on the hardware used to display an image. If you test same software on different PCs, the bitmaps may completely fail to match when the images are clearly the same.

- Aim for a good balance between sensitive and robust tests: The breadth tests that will always run every time anything changes, should be relatively few in number but mainly sensitive to any changes. Where depth tests that each explore a particular area of feature in detail but are generally only run when an area has been affected by change, aim for robust tests.

### Daily-Test tool and Comparison

According to Bergdahl [8] the Daily-Test tool performs comparison by comparing an XML file contains the actual result with another XML file that contains the expected result that is specified by the user. It is a simple, dynamic comparison technique that fits for the purpose of comparison that MPC want to achieve with the technique so far. But in the future if there is a need to use other comparison techniques described above, then the guidelines for effective and efficient comparison should be used in order to choose and implement the right comparison technique for MPC.

The reason why we recommend guidelines instead of proposing other comparison techniques is that there are many decisions to be made throughout design and implementation of comparison that will depend on different scenario of the current situation. For example:

- When to compare as much as possible as often as possible then a sensitive approach will be the right choice.
- When to compare minimum of information, robust comparison will be needed.
- When to compare actual and expected outcomes with known differences, then a complex comparison will be the right choice
- And where there is a need for identical match between the actual and expected outcomes, simple comparison is the right way to go.

Therefore, recommending a specific comparison technique might not be valid for different situations. Guidelines for how to choose efficient and effective comparison will probably help to make the right choice for a comparison technique for different situations. This in turn may allow you to combine different approaches in order to achieve an efficient and effective comparison.

## Conclusion

Automated testing techniques described in this chapter were scripting and comparison. The two most common techniques that are needed to make test automation easier. The use of scripts is an important issue in test automation mainly for the reason where scripts make test automation easier. But one of the problems with scripts is how to deal with the maintainability of test scripts. This is a problem that related one of the objectives that MPC want to achieve when using automated testing. MPC had maintainability problems with artefacts used to automate tests. Those artefacts were not reusable and in some cases, they have been throwing out between MPC versions. Therefore, different scripting techniques that offer a way to improve maintainability of scripts were described and evaluated by using selection criteria in order to decide which technique that is appropriate for MPC. We believe that Keyword-driven scripting technique is appropriate for MPC as it allows creating maintainable and structured test set and automating its execution in a reusable way. Other reasons why this technique is most suited for MPC were also described in this chapter.

Comparison techniques are used to verify if the software produce the correct outcome where comparison between an actual outcome and expected outcome of the test will be performed. This comparison could be done manually but in some cases such as verifying repetitive and detailed tasks could be better to automate because these tasks are boring and it is easy to make mistake if performed manually. The comparison technique that is in use within MPC is dynamic simple comparison where xml files that contain the actual outcome and the expected outcome of the test is compared and depend on whether or not the files compared are the same or not, the term 'Test report - OK' or 'Test report - n error (s)' are used to indicate respective case. In this chapter, we did not select an specific comparison technique for MPC instead we recommended guidelines for effective and efficient comparison in order to be able to combine different comparison techniques that helps one to implement an effective and efficient comparison technique.

Test analysis that will be performed manually after the test execution can take a considerable amount of time if they are not organised in such a way that makes them easy to go through and find the bugs founded by the comparator. We believe that one should develop a logical hierarchy of subsets where the first level subset will contain the most general tests and the lower levels of subset will contain the more detailed tests. Having such logical hierarchy will help you to minimise the time for result analysis where you don't need to spend time to analyse in the lower level if any tests in the first subset level fails because faults in the first subset level may cause other tests to fail in the lower level subsets.

Chapter 7

# Conclusions

We have concluded that MPC deals with problems that are common among organizations that use automated testing for their products. To address these problems, we decided on two objectives for automated testing at MPC together with Daniel Bergdahl; efficiency and maintainability. With those two objectives in mind we identified what to automate, which techniques to use and improvements related to their process.

The actual selection of test cases affects the ability to reach the test automation objectives to a higher degree rather than in which test level you chose to automate. The different characteristics of test cases decide whether or not they are suited for automation. Test cases dealing with huge amount of traffic such as stress testing, performance testing and capacity testing are suited for automation because they are resource demanding. Regression testing is also suited for automation because it is often performed frequently as changes are made to the system. Repeatable tests tend to be boring and time consuming to do manually and tools are better to perform such activities. A careful analysis of system requirements and functions is needed in order to decide on which parts are most suitable for automation.

By preparing for test automation it will be easier to succeed with the test automation effort. One way of doing this is to design for testability, with test automation in mind. A system designed for testability will lay the foundation for test automation engineers to more easily automate beneficial tests. With the results of the analysis described above it is possible to identify the parts of the design that automated testing is depending on. Another important issue regarding the success of test automation is the communication of test automation objectives. It is necessary to decide what are the objectives of test automation before trying to implement it. Expectations and actual results will most likely not be the same if everyone involved (involved in what?) is not working towards the same goal.

When the test automation preparations are done it is time for actual implementation of test cases. The most common techniques used when automating tests are scripting and comparison. We have found that keyword driven scripting is a suitable technique for MPC because it was the most maintainable scripting technique of the ones that we evaluated. The comparison technique is used to verify the output of an automated test against the expected output. The current test automation tool at MPC use a dynamic comparison where whole files are compared against each other. We found the current comparison technique at MPC fits to its purpose so far and could not make any improvement suggestions. Instead we included comparison guidelines to be used when adding new comparisons to test tools in the future. Manual test analysis that is performed after the test execution can take a considerable amount of time if the results are not well organised. We believe that a test result viewer that displays test results in a logical hierarchy of test result subsets should be developed. If tests in higher-level subset fail then you do not need to spend time to analyse the lower-level subset because fault in the higher-level subset may cause other tests in the lower levels to fail.

We have shown what can be done to improve the maintainability and efficiency of automated testing but is it possible that automated testing can reduce the cost of testing? The costs of manual and automated testing can be quantified respectively

but a comparison is harder to do because they are not interchangeable. In the same sense it is difficult to quantify the benefits of automated testing compared to manual testing. Quantification of automated testing benefits is very hard to do since there are no clear definitions in economical term. This makes it hard to know if test automation can decrease the cost of testing. We believe that the test automation effort must be made with a strategy and in the long run, as any immediate payback is eaten up by the initial costs. Focus should not be on reducing the test budget but rather prevent unexpected quality costs. Test automation can help testers in their pursuit of finding defects in software and we believe this is the most important aspect of test automation, not as a mean to save money.

We believe that test automation, if used correctly, can be an instrument to either get greater test coverage or shorten the amount of actual testing. If test automation is developed in parallel with the product it is supposed to test, we believe this will give test managers the ability to efficiently test certain parts of the products which will save them time and effort. This benefit can be used to either shorten the testing time or be used to test parts of the product not yet covered by testing. This does not mean that test automation will reduce costs directly but will help to shorten the time to market.

Appendix 1

# Further Research

## Introduction

This is a short chapter where we present three topics for future research that we discovered during our work with this paper.

## Cost reduction of testing

Our analysis of test automations effect on costs was limited to a discussion about literature findings. A deeper and more practical analysis with a more economical focus might reveal the real effect test automation has on the cost of testing. As we have shown there are several issues that need to be resolved in order to quantify the benefits in terms of money. We believe this would make a good topic for further research not only from a software engineering perspective but also from a business economics perspective. An alternate focus might be to find out what can be done to reduce the cost of testing in general, without focusing only on test automation.

## Test Automation Introduction Process

During our studies we have found several issues that relate to the problem of introducing test automation to a test organization. Some of these issues did not apply for the situation at MPC because they already had a test tool in place. However, there are many problems to overcome and we would like to see a paper that describes the pitfalls of introducing automated testing to an organization and how to overcome them. This would be of great help for test managers and others that want to make use of the benefits offered by automated testing.

## How to design for testability

We identified design for testability as an important tool for achieving the test automation objectives. Our scope did not allow us to go into the details of how to design for testability. There are some short papers written on this subject that we have found but they are only touching the surface. We believe this could be a matter of further research. Either by looking specifically at how a certain product can be redesigned for testability or finding general design rules for testability.

## Test engineer training

Testing is an important part of the development process but how seriously does companies really take it? Are the testers really skilled in testing? Does training of the test engineers improve the quality of system under test? Is there a relationship between the test skill and the effort/schedule? It would be interesting to find out if this kind of training saves money or not.

Appendix 2

# References


[1] Elfriede Dustin, Jeff Rashka and John Paul (Automated Software Testing, 1999)

[2] Daniel Bergdahl (Thesis Proposal Simulators)

[3] Douglas Hoffman (Cost Benefits Analysis of Test Automation, 1999)

[4] Christian W. Dawson (The Essence of Computing Projects – A Student's Guide, 2000)

[5] Tim Koomen and Martin Pol (Test Process Improvement, 1999)

[6] John Watkins (Testing IT, 2001)

[7] Mark Fewster & Dorothy Graham (Software Test Automation, 1999)

[8] Daniel Bergdahl, (Technical description Component FSC-Daily Test, 2001)

[9] Lars-Johan Ljung (Main Test Plan, Serving Mobile Positioning Centre 5.0)

[10] I D Hicks, G J South and A O Oshisanwo (Automated testing as an aid to systems integration)

[11] Kepple L R: 'The blackart of GUI testing', Dr Dobb's Journal (1994)

[12] Bill Boehmer, Bea Patterson (Software Test Automation-Developing an Infrastructure Design for Success)

[13] Brain Marick (When Should a Test Be Automated)

[14] Linda G. Hayes (The Automated Testing Handbook, 1995)

[15] GSM 03.71: "Digital cellular telecommunications system (Phase 2+); Location Services (LCS)"

[16] http://inside.ericsson.se/ttm/index.html

[17] Peter Carlsson (Test Plan – Unit Basic and System Desing Test, Serving Mobile Positioning Center 5.0)

[18] Jan Fex, Richard Downey (Basic Test Plan for PPLocator)

[19] Bret Pettichord (Seven Steps To Test Automation Success, 1999)

[20] Elisabeth Hendrickson (Evaluating Tools, STQE Magazine, 1999)

[21] Elisabeth Hendrickson (Buy it or build it?, STQE Magazine 2000)

[22] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries, 1990

[23] Elisabeth Hendrickson (Bang for the buck test automation, Test Automation Spring 2001)

[24] Magnus Jönsson (Sub-Process Component Coding + BT (Basic Test), http://inside.epk.ericsson.se/iwdocs/49645894.ppt)

[25] James Bach (Test Automation Snake Oil, 1999)

[26] Douglas Hoffman, (Cost Benefits Analysis of Test Automation, 1999)

[27] Linda G. Hayes (Does Test Automation Save Time and Money?, 2001)

[28] Hans Buwalda and Maartje Kasdorp (Getting Automated Testing Under Control, 1999)

[29] http://xprogramming.com/software.htm

[30] Kerry Zallar (Automated Software Testing – A Perspective)

[31] Bret Pettichord (Design For Testability, Software Test Automation Conference, 2001)