M A S T E R  T H E S I S:

T E S T I N G  T H E  P R O D U C T  P R O P A G A T I O N

- S Y S T E M  T E S T I N G  U S I N G  T H E  B E N E F I T S  O F  P R O D U C T  L I N E S -

Authors:
Frida Nordlund, cii00fn8 @ i.lth.se
Stefan Åshäll, d01sas @ efd.lth.se

Supervisors:
Carina Andersson, LTH
Ulf Rexelius, SEMC

Assigner
Per Runeson, LTH
Fredrik Töörn, SEMC
Abstract

Product line development is a successful approach to software re-use and can save a large amount of project resources when applied. At SEMC a product line strategy was introduced in 2002. Since the introduction resource saving has been successful mainly during the development phase, leaving the testing phase an important area in which all the benefits from the product line architecture needs to be further exploited.

The purpose of this Master Thesis was to explore the advantages of product lines during system testing. The goal was to produce a testing strategy that optimizes the system testing used at SEMC by taking full advantage of the benefits of product lines.

First of all, a pre-study was made of how software is developed at SEMC, with a special focus on testing and the product line architecture. Focus group meetings were then held in order to identify and prioritize areas in which testing needed to be improved. A study was made based on these improvement areas and a product line test strategy was formulated.

The three most important improvement areas were the following: keeping track of differences between products; selecting the right test cases to run; creating exact and complete test specifications. Product differences are used to define the test object and by selecting test cases that exercise all of the code in the test object, testing can be made more efficient. If the test cases are complete and exact these three improvement areas can be used to optimize testing.

The results from the study have been formulated into a system testing strategy that can be used when working with software product lines. The strategy is based on the most important improvement areas and shows how the combination of these can improve testing in a product line environment.
Foreword

Conducting this Master Thesis has been an exciting journey that begun in September 2005 and ended in February 2006. The initial assignment was to compare the return rate to the amount of system testing performed on products in one specific product family. The underlying goal was to optimize system testing of product families. The initial assignment was eventually transformed into development of a System Test Strategy that takes advantage of the benefits of products lines at a company with a growing product portfolio.

The reactions on the Master Thesis project at Sony Ericsson Mobile Communication (SEMC) have been very positive. The Master Thesis emphasised the need of monitoring variability between products in the product line and has led to the creation of a tool at SEMC that will be used to monitor product variability.

The reactions on the Master Thesis project at LTH have also been very positive and the Master Thesis report will be used as a pre-study for a research project which is a combined effort between SEMC and the department of Telecommunications at LTH. The research project will begin in March 2006 and continue for three years.

We feel that conducting the Master Thesis project at SEMC have given us great insight in pros and cons of product line development in a large scale embedded system environment. It has been an opportunity to learn both how complex software products are developed and what the effect software architectures have on the development process.

We hope that the recipients of this Master Thesis Project will find its conclusion useful and the presentation enjoyable.
Acknowledgements

We would like to thank our advisor at Lund’s Institute of Technology, Carina Andersson, for her enormous support during the creation of this Master Thesis. We would also like to thank Per Runeson, Mikael Christersson and Fredrik Töörn for making this Master Thesis project possible.

A large amount of people have helped us to finish the Master Thesis. We wish to thank all of the following people:

Andersson, Carina  
Andersson, Michael  
Bjorne, Frederik  
Bäck, Magnus  
Carlsson, Lenny  
Christersson, Mikael  
De Silva, Peter  
Fänge, Thomas  
Hedberg, Anders  
Jonsson, Göran  
Karlsson, Even-André  
Källestål, Christina  
Lacayo, Eric  
Levin, Mattias  
Mats, Lars  
Mikkelsen, Max  
Mossberg, Niklas  
Möller, Rickard  
Nilsson, Anders  
Nilsson, Magnus  
Nyberg, Anders  
Olofsson, Jerker  
Persson, Pernilla  
Pettersson, Roger  
Rexelius, Ulf  
Runeson, Per  
Sandahl, Christer  
Svensson, Peter  
Tomasson, Anna  
Töörn, Fredrik

You made it happen!
# Index

1 **Introduction** .................................................................11
   1.1 Background ...............................................................11
   1.2 Problem definition ......................................................12
   1.3 Goal & Purpose ............................................................12
   1.4 Outline .................................................................13

2 **Theory** .........................................................................15
   2.1 Embedded systems .......................................................15
   2.2 Software Engineering ...................................................15
   2.3 Software measurement ..................................................24

3 **Method** .......................................................................29
   3.1 Work process ............................................................29
   3.2 Scope .................................................................31

4 **Pre-study at SEMC** .......................................................33
   4.1 Product family description ............................................33
   4.2 Product lines at SEMC ................................................35
   4.3 Configuration management at SEMC ...............................35
   4.4 Test process ............................................................36

5 **Results** .......................................................................41
   5.1 Product variability - Software ........................................43
   5.2 Product variability - Hardware ........................................49
   5.3 Product variability - File system .....................................50
   5.4 Product variability - Signalling platform .........................50
   5.5 Product variability - Marketing .......................................51
   5.6 Product variability - Configuration ................................52
   5.7 Test case selection .....................................................53
   5.8 Specifications ............................................................57
   5.9 Product family planning ...............................................61
   5.10 External development and testing ...................................62
   5.11 Operator variability ...................................................63
   5.12 Sixth sense decisions ................................................63

6 **Conclusions** ..............................................................65
   6.1 System Test Strategy ....................................................66
   6.2 Evaluation of strategy ................................................69
   6.3 Generalization ..........................................................71

7 **Future Work** ................................................................73
   7.1 Product family planning ..............................................73
   7.2 More product lines ......................................................73
   7.3 Improve encapsulation ................................................73

**Reference list** ................................................................75

**Appendices** ..................................................................79
A Questions and metrics ....................................................79
B Word Index ....................................................................80
CHAPTER 1: INTRODUCTION

CONTENTS:
- Background
- Goal & Purpose
- Problem definition
- Outline
1 Introduction

The first mobile phones were developed, by Dr Martin Cooper general manager at Motorola’s system department, in 1973\(^1\). Since then mobile phones have evolved to include much more functionality than just setting up a phone call between two parties. Today mobile phones include a wide range of functionality such as calendars, alarms, internet browsers, e-mail clients, games, radios, and media players. Mobile phones have become mobile handsets. Mobile handsets consist of a vast amount of software and they have microprocessors with very high performance; they are embedded systems. The evolution from mobile phones to mobile handsets has lead to an increase in functionality and complexity, which greatly affects development and testing; new ways to develop and test mobile handsets are needed.

The mobile handset market is fast-paced and dynamic with high demands on the products; they should be high-tech, easy-to-use and cheap. Tough competition within the market has increased the demands on companies producing mobile handsets. Providing a wide variety of products at a low price is the only way to survive in this new, growing market. The average life-time of a mobile handset ranges from three months to one year; new products need to be developed fast, yet with high quality.

Sony Ericsson Mobile Communications (SEMC) is the result of a joint-venture between the Japanese electronics giant Sony Corporation and the Swedish telecommunications company Ericsson. It was established in 2001 and is today a major actor on the mobile handset market; SEMC has about 7% of the overall market share. SEMC has in total over 5000 employees, based all around the world: in Sweden, Japan, Germany, China, and the United States. Global management is located in London.

1.1 Background

Software development is a process focused on producing products that comply with demands from customers and end-users. A part of this process is software testing which aims to ensure that these demands are met. Therefore, the process used to test software is extremely important and one major factor when ensuring the quality of the final product.

Software test processes can be somewhat complicated; they generally involve a large amount of people and many different organizations. Therefore it is vital to have a structured approach to testing where specification, execution and documentation of tests are done in a predefined manner. Having a well-functioning test process is the basis of ensuring that products are well-functioning and complete the day they are released on the market.

Software testing is an expensive part of the software development process. Since testing is the last phase through which a product passes, it is also the company’s last chance to ensure that their product will be delivered with a satisfactory level of quality. Therefore, a lot of resources are allocated to testing and the main problem in software testing today is how to minimize costs without sacrificing testing quality.

\(^1\) Nate Orenstam, http://www.valleyofthegeeks.com/Features/Cooper.html
1.2 Problem definition

There are several important reasons why a product line is a working and efficient architectural concept for software systems. In general, it enables the production of many different products at a relatively low overall price. With more products a larger part of the market can be covered; the products can be adjusted to suit specific customer group’s needs. Thus, the future calls for product lines containing more products, to cover larger parts of the market while maintaining or lowering production costs.

To be successful when working with product lines there is a need for strategies; optimize planning, development and testing to exploit the benefits of software re-use to the maximum. Since product line is a relatively new paradigm in Software Engineering these strategies are still under development.

1.3 Goal & Purpose

The goal of this Master Thesis is to create a testing strategy that optimizes the way system testing is performed at SEMC by taking full advantage of the benefits of product line. The report has a focus on improving testing at SEMC as well as contributing to research in the field of Software Engineering.

The purpose of this Master Thesis is to explore the advantages of product lines during system testing. This is done by combining the knowledge gained from a study of how SEMC tests their products today and by studying relevant, contemporary research from the field of Software Engineering. Prior problems using product lines during system testing at SEMC is analyzed in order to focus the evaluation on areas that can be improved.
1.4 Outline

Figure 1 shows the general outline of the report, the seven main chapters and their contents. The first chapter gives an introduction to the mobile handset market, a background to testing and finally the purpose, goal and problem of the Master Thesis project are presented. Chapter two contains important theoretical knowledge needed to understand this thesis. It gives an introduction to Software Engineering, Embedded systems and the Goal/Question/Metric method used to perform the Master Thesis. The third chapter gives an overview of the work process used in the Master Thesis.

The fourth and the fifth chapters present the results from the GQM study made at SEMC. The fourth chapter presents the general study of SEMC while the fifth chapter presents the results from the study of the improvement areas defined by the GQM study. The sixth chapter presents the conclusions derived from performing the GQM study: a system test strategy and how the conclusions can be generalized. Finally, chapter seven presents three suggestions for future work.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Background</td>
<td>Embedded systems</td>
<td>Work process</td>
<td>Product family description</td>
<td>Software Variability</td>
<td>System Test Strategy</td>
<td>Product family planning</td>
</tr>
<tr>
<td>Problem definition</td>
<td>Software Engineering</td>
<td>Scope</td>
<td>Product lines at SEMC</td>
<td>Hardware Variability</td>
<td>Generalization</td>
<td>More product lines</td>
</tr>
<tr>
<td>Goal &amp; Purpose</td>
<td>Software measurement</td>
<td></td>
<td>Configuration management</td>
<td>File-system Variability</td>
<td>Improve encapsulation</td>
<td>Improve encapsulation</td>
</tr>
</tbody>
</table>

Figure 1. Outline of the report.
CHAPTER 2:
THEORY

CONTENTS:

Embedded systems
Software Engineering
Software Metrics
2 Theory

This chapter gives an introduction to the Software Engineering discipline and the software product line concept is introduced in that context. The chapter aims at giving a theoretical background to the Master Thesis. It explains important terms, such as embedded systems, product lines, system testing and the Goal/Question/Metric method.

2.1 Embedded systems

A mobile handset is an embedded real-time system that must react to events generated by the hardware and issue control signals in response to these events. It is embedded in a hardware system and must respond in real-time to events in the system’s environment. One definition of a real-time system is: "a system whose correctness includes its response time as well as its functional correctness."

Real-time systems need to not only function correctly but to do so within a specified time limit. Real-time events that affect the system may be triggered by user input or time-triggered events that the system itself manages. Embedded real-time systems can be found in many different products e.g. refrigerators, wrist watches, cars or industrial robots.

2.2 Software Engineering

Software engineering is a computer science discipline concerned with developing large systems. Software engineering covers not only the technical aspects of building software systems, but also management issues such as directing programming teams, scheduling and budgeting.

The goal of software engineering is to produce software with high quality at a reasonable cost. High quality in this context is defined as software that is maintainable, dependable, efficient, and usable. The development costs are usually divided into: specification, design and implementation, validation and verification, and evolution. Normally integration and testing are the bigger parts. They can sometimes be close to 50 percent of the overall software development cost. To develop software of high quality you need to have a number of well defined processes. This chapter briefly defines the processes that are most essential to this Master Thesis.

2.2.1 Software Engineering Process

The software engineering process is a structured approach that describes the activities that result in a completed product. A number of generic process models exist, such as waterfall, spiral, and iterative.

---

2 Douglass Locke, http://www.linuxdevices.com/articles/AT6090565653.html
The fundamental phases are the same in all these processes: *software specification, design and implementation, validation and verification, and evolution*. The generic software development process is shown in Figure 2, above. It shows an iterative process where the output from each phase is fed both back to the previous phase and forward to the following phase.

**Software Specification**
The *specification* phase defines the software functionality by establishing system requirements in a requirements specification. The software specification is the most critical phase of the software engineering process since problems that are not resolved at this phase inevitably will lead to problems in the design and implementation phases. When all the requirements are collected a classification of the requirements is made into coherent clusters, and then prioritized. Finally the requirements are validated to discover if they are complete, consistent and in accordance with what stakeholders want from the system. The elicitation and analysis of requirements is an iterative process; validation often generates changes in the requirements which forces elicitation to be re-made.

**Software design and implementation**
The *design and implementation* phase is concerned with designing according to the specification and implementing the system according to the design chosen. The later part of the design is interleaved with the implementation and that is why design and implementation is stated as one activity.

**Software Validation and Verification**
The *software verification and validation* phase ensures that the system meets the specification and the expectations of the customer. Verification is making sure that the product meets the requirements. Validation is making sure that the product is functioning the way the customer wants.

---

Figure 3 shows a five-stage testing process. Large software systems are not tested as single units. They are built from units that are integrated into modules which in turn are composed into sub-systems. The testing process should therefore proceed in stages where testing is carried out incrementally. Ideally, component faults should be discovered early in the process and interface problems when the system is integrated. As errors are discovered previous stages in the testing process must be repeated e.g. when errors are found they have to be corrected and the correction tested through the different test stages. As seen in Figure 3 the process is iterative with information being fed back from later to earlier stages of the process.

Here follows a detailed description of the different stages of software testing.

- **Unit testing**: the individual components of the system are tested separately to ensure that they are operating correctly. Units are normally testable pieces of code, such as functions (in C) or methods (in Java).
- **Module testing**: the module, which is a collection of dependent components such as functions, is tested. An example of a module could be SMS.
- **Sub-system testing**: a collection of modules that have been integrated into a sub-system is tested during sub-system testing. The sub-system test focuses on the detection of module interface faults by rigorously exercising these areas. One example of a sub-system could be Messaging which could include modules such as SMS, MMS, e-mail, etc.
- **System testing**: the testing of the complete, integrated system is called system testing. In an embedded system this includes both software and hardware. The system test is concentrated on finding errors that result from interactions between sub-systems and the hardware. It is also concerned with validating that the system meets its requirements.
- **Acceptance testing**: the final phase before the system is accepted for operational use is acceptance testing. During acceptance testing customers or end-users test the system in order to verify that it meets their demands. For example, using the system in a real environment with real data may cause errors that would not have
been found during system testing. Acceptance testing often reveals errors in the system requirements, for example if the system does not meet the users’ needs or when system performance is unacceptable low.

Software Evolution and Maintenance
Software engineering is a creative process where a software system is developed from an initial concept through several steps into a working system. Software evolution and maintenance is the process of re-using the system once it has gone into use. Very few software projects today start with a completely new system; most often software is re-used between projects. Therefore it makes more sense to view development and maintenance as an evolution. Figure 3 shows how software often is re-used between projects over time. The previous systems are used as a base for the new system.

![Figure 4. Functionality is added over time, the new system is based on the old systems’ software.](image)

2.2.2 Software Testing
Software testing involves both software validation and verification. The aim of software testing is to ensure that a system or a product meets customers and end-users demands. In this chapter software testing is explained with a special focus on test development and test techniques that can be used during integration testing of embedded systems.

Test Development
A test case is a description of an action that exercises the system in some predefined manner. It is usually divided into a series of steps each with one expected result or outcome. Larger test cases may also contain prerequisite states or steps, and descriptions. The test cases are usually stored in a word processor document, spreadsheet, database or other common repository. In a database system, you may be able to see past test results and who generated the results and the system configuration used to generate those results.

Integration test cases are developed based on the system’s requirements. Ideally there should be a one-to-one correlation between a test cases and its requirement. This enables the tester to easily verify that the system is working correctly by making sure that the product meets the requirements. The test cases are grouped and collected into a test specification. Different test specifications can be divided into test suites; configured for special types of testing. Some test suites are used to test large parts of the system while others can be more specific testing only a small part of the system thoroughly. Different test suites are run at different times in a project. The information about when a specific test suite should be run during a project is often collected in the test plan.
Integration testing
Integration testing is performed when the individual components of the system has been component tested and are integrated to a complete system. Integration testing focuses on testing the interfaces between the components. Both module testing, sub-system, and system testing can be seen as integration testing. The testing referred to here is the integration testing which takes place when the components have been integrated into a complete system i.e. system testing.

System Testing
System testing is testing that is conducted on the complete, integrated system to evaluate the system’s compliance with its requirements. System testing is generally based on black-box testing techniques. In black-box testing the internal workings of the test object are not known and the tester focuses mostly on how the system reacts to different inputs. This is opposed to white-box testing which studies and tests different parts of the system, in detail. System testing tends to be more of an investigatory testing phase, where testers tend to have an almost destructive attitude and not only test the design, but also the behaviour and the believed expectations of the customer. System testing is intended to test up to and beyond the software and hardware requirements specifications.

As software faults are found during system testing new software builds are released that include corrections of detected faults. The incremental nature of system testing is controlled by defining regression tests.

Regression Testing
Regression testing is any type of software testing which seeks to uncover regression errors. Regression faults occur whenever software functionality that previously worked as desired stops working or no longer works in the same way that was previously planned. Typically regression errors occur as an unintended consequence of program changes such as error corrections. Common methods of regression testing include re-running previously executed system tests and checking whether previously fixed errors have re-emerged. When building large and complex systems regression testing consumes a large part of the organizations’ software development budget. The building phase of an information system takes on average about 20 percent of the system’s lifecycle. During the reminding 80 percent of the system’s lifecycle regression testing is needed in order to ensure that system quality is not deteriorating due to error corrections. If regression testing is performed manually, the execution of test cases may be one of the most expensive activities of the testing process. Therefore it is of interest to reduce time and resources devoted to regression test execution. When only limited changes have been made to the software, a complete re-run of all test cases may be unnecessary and instead executing only a subset of the full test suite may suffice. To select test cases for re-execution is referred to as regression test selection. One way of focusing regression testing is to focus testing on the changes that have been inserted in the system since the last time testing was performed. This is referred to as change-based test selection.

---

6 Software builds are described in Chapter 2.2.3.
However, using change-based test selection for larger systems can be complicated. For small systems developers can easily communicate where changes have been inserted and those changed areas can be thoroughly tested. In larger systems this hands-on approach is not possible and more structured methods are needed. There is a need to monitor changes to the system but also to know how changes affect the test object, i.e. there is a need to know which test cases exercise which functions. One method of gathering information on how test cases exercise the system is called coverage analysis.

Coverage analysis, a white-box testing technique

Code coverage analysis is a method used for white-box testing. It describes the degree to which the source code of a program has been tested. It is different from black-box testing methods because it focuses on the code directly. Code coverage techniques were amongst the first techniques invented for systematic software testing. There are a number of different ways of measuring code coverage, the main ones being:

- **Statement Coverage** - Has each line of the source code been executed and tested?
- **Condition Coverage** - Has each evaluation point (such as a true/false decision) been executed and tested?
- **Path Coverage** - Has every possible route through a given part of the code been executed and tested?
- **Function coverage** – Has every function in the code been exercised?

Full code coverage is usually impractical or impossible. Any module with a succession of \( n \) decisions in it can have up to \( 2^n \) paths within it; loop constructs may result in an infinite number of paths. The resulting output is analyzed to see what areas of code have and have not been exercised. Code coverage is ultimately expressed as a percentage, as in "We have tested 67% of the code". The meaning of this depends on what form of code coverage have been used, as 67% path coverage is more comprehensive than 67% function coverage.

Extracting code coverage information using different tools and libraries often reduces performance of the test object. This is unacceptable when performing normal operations of the software. There are also some defects which are affected by such coverage tools. In particular some real-time sensitive operations are impossible to detect while run under code coverage environments; and conversely some of these defects are only triggered as a result of the additional overhead of the coverage analysis tools.

An important issue when using code coverage analysis is the questions of whether 100% code coverage really means that a product is thoroughly tested. It is important to realize that only faults that occur because of the code that is already written can be found using code coverage. However, most often faults occur because of code that has not been written (so called faults of omission). Setting goals for code coverage percentage, without taking other factors into account can be dangerous and inefficient. However, using code coverage information together with other statistics can be very useful, especially when optimizing regression test selection.

---

Test Selection
Test selection needs to be done using a safe and precise technique to select test cases. These two criteria have to be taken into account when selecting a test selection strategy:

Criterion 1 – Safety
A safe technique is a technique that under given circumstances select test cases that will expose the same set of faults as the complete test suit would expose. A safe test selection technique will consider the effects of new and added functionality as well as modified functionality.

Criterion 2 – Precision
The retest all strategy is safe but it is also imprecise. In addition to selecting all potentially revealing tests, it also selects tests that cannot possibly exhibit changed behaviour. A test selection strategy with good precision selects only the test cases that exercise areas that need to be tested. Therefore we seek to maximise precision.

Figure 5 shows a graphical representation of a safe and precise test selection technique. The black circle represents the original test suit and red crosses are errors. The smaller, dotted circle shows an example of a safe and precise test selection strategy.

```
Figure 5. A graphical description of a safe and precise test selection strategy.
```

2.2.3 Configuration Management

Configuration management is the development of standards and procedures used to manage the software of an evolving software product. Configuration management procedures define how to record software changes, how to relate these to software components and the methods used to identify different versions of the software. Developers submit their code, changed or new, and configuration management receives it, integrates it with existing code and finally releases a software version, when a complete product has been implemented. Configuration management tools are used to store versions of software components, build software system from these components and track the release of software system builds9. Creating the final software version is called building and therefore software versions are referred to as builds.

Making new builds often increases the chances of finding problems that stem from component interactions early in the process. Daily builds encourage thorough unit testing since developers are put under pressure to not deliver components that cause the whole

---

system to fail. Finding errors early in the test process is to prefer since errors tend to cost more to correct when they are found in later phases of the development.

2.2.4 Software Architecture and Design Process

Large software systems are usually decomposed into smaller subsystems to provide a good overview and display internal dependencies. Architectural design is the process of identifying sub-systems and establishing a framework for control and communication. The output of this process is the **software architecture** and it has three major purposes:

- A high level representation of the system is useful when discussing with stakeholders.
- It allows an early analysis of the system which facilitates verifying performance and reliability requirements.
- It supports large-scale re-use since using a software architecture enables identification of re-usable components. An architecture can also be transferred to other systems with similar requirements.

The last one of the three purposes, software re-use, has been an important goal in Software Engineering since research began in the field. Re-using software between projects results in major savings; the succeeding project begins working with an already well-functioning product. Software re-use has traditionally been implemented between projects in time; today an emerging method for software re-use is called product **line development**.

**Product line development**

The concept of product lines has been adapted as a successful approach to Software Architecture: it enables the development of many different products, at a fast rate, with high quality while maintaining a reasonable price. The concept of product lines (or **product families**) is based on having a set of common components and from those components define individual products. By specifying, developing and testing the whole set of components it is possible to produce several different products with high quality at a low cost. Different products in the same product line can easily be adapted to suit specific market segments at a low cost.

---

12 The terms **product line** and **product family** will both be used in this report. The term **product line** will be used to describe the idea behind product lines while **product family** will be the noun that describes a group of products that have been formed into a product line. The full set of all products produced by company (in this case SEMC) will be referred to as the **product portfolio**.
The example in Figure 6, taken from a company that produces different types of servers, shows how similar products can be grouped into a product line. Functionality has been grouped so that all servers include file-system functionality and a common interface towards network protocols, since these two components are equal for all types of servers. Common functionality can also be grouped into sub-groups that inherit functionality not only from the complete product line but from the sub-group as well. This type of sub-groups is exemplified in Figure 6 as storage-servers and printer-serves where a CD-ROM-server shares functionality with all storage-servers as well as with the whole product family.

Variations of these server types are easily produced, making it possible to modify them to suit specific needs. The result is that the company can offer a wide variety of servers without having huge development costs. In the example in Figure 6, a DVD-ROM-server could be added without having to implement new network protocols, file-system functionality or storage-server functionality, since these parts have already been implemented for existing products.

As seen above, product lines offer increased flexibility at a low price. As long as the development process ensures a high level of quality for all common components within the product family, the finished products will most likely be of equally high quality. Another important factor is maintenance cost, which can be considerably reduced if the system is component-based and products share a majority of the functionality. There are however several differences between the theoretical and industrial view on product lines and re-usable components\textsuperscript{13}. These differences have several explanations and as in most research fields practice and theory do not completely match.

\textsuperscript{13} Jan Bosch, \textit{Design & Use of Software Architectures}, Pearson Education, Edinburgh, 2000
Table 1. Differences between theory and practice in using product lines.

<table>
<thead>
<tr>
<th>Theory</th>
<th>Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>The product line architecture is explicitly defined and described.</td>
<td>Mostly conceptual understanding of product architecture. Definitions passed on through simple notations and sketches.</td>
</tr>
<tr>
<td>Architecture consists of individual components.</td>
<td>Components are not well divided. They are “made to fit” using workarounds and adapted between system assets.</td>
</tr>
<tr>
<td>Architectural Design Languages (ADLs) are used to automatically generate applications.</td>
<td>More complex programming languages such as C, C++ and Make are used to describe the system configuration.</td>
</tr>
<tr>
<td>Re-usable assets are black box components and therefore easy to re-use in different architectures.</td>
<td>Assets are large pieces of software with a complex internal structure and no enforced encapsulation.</td>
</tr>
<tr>
<td>Assets have few and explicitly defined variation points.</td>
<td>Variation is implemented through configuration and specialization or replacement of parts of the assets.</td>
</tr>
</tbody>
</table>

As seen in Table 1, implementing a product line architecture can be complicated, especially when the product line consists of complex products with a large amount of varying requirements. Bosch\textsuperscript{14} proposes a way to define maturity in product lines and gives examples of difficulties that are common for companies that are adapting to a product line architecture. One common problem is to manage variability. Variability describes the extent to which the products that form the product lines differ.

### 2.3 Software measurement

Software measurement is concerned with deriving a numeric value for some attribute of a software product or software process. By comparing these values to each other and to other standards which apply across an organisation, it is possible to draw conclusions about the quality of the software or software processes. These attributes are called metrics and are often used as the basis for evaluations and studies of software development. Software metrics are data that are interesting when evaluating or analyzing software projects. Metrics can be used to make predictions for future projects, measuring product quality and calculating the complexity of a product.

Examples of metrics from a test project could be\textsuperscript{15}:

- **M1. Cost of testing**
- **M2. Number of faults found**
- **M3. Average cost of each fault = cost / number of faults found**

As seen in the examples above, various types of metrics exist, some directly measurable (M1 and M2), some defined by other metrics (M3). An important distinction is the difference

\textsuperscript{14} J. Bosch, *Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization*, 2002

between metrics derived from opinions and feelings (from now on referred to as *soft metrics*) and metrics derived from actual facts.

### 2.3.1 The Goal/Question/Metric Method

Goal/Question/Metric (GQM) is a method used to organize software measurements. There are several ways to organize and control software measurements. The GQM method guides the choice of which metrics to collect, before the data extraction begins. This way no unnecessary work is spent on collecting metrics that turn out to be irrelevant for the study. GQM provides a clear and traceable answer to why each metric should be collected. The purpose of the data extraction is therefore never lost since GQM provides a method to collect individual metrics while keeping the focus on the goal of the data extraction. A Master Thesis performed at Hewlett-Packard shows how GQM can be used to study software processes.\(^\text{16}\)

GQM is based on two main ideas\(^\text{17}\): (1) analyses of software processes should be goal-based rather than metrics-based, and (2) goals and metrics that are to be collected need to be adapted to the object of study. An analysis should be focused on why it is being made. A focus on just the collected metrics will lead to unnecessary data collection and a less organized data analysis.

![Figure 7. The GQM process, simplified](image)

### 2.3.2 First step: Goal definition

As seen in Figure 7 the first step when working with GQM is to define the goals of the study to be performed. Typical goals for analyzing software processes are\(^\text{17}\): increase quality, shorten project cycle time, decrease costs and decrease risks. When defining a GQM goal it is important to also specify *purpose* (what object to study and why), *perspective* (what aspect and with which viewpoint) and in what *context* the study will be made. There are detailed templates available that clearly define how a goal can be formulated. Table 2 shows an example of a GQM goal aimed at increasing the quality of a test process.

<table>
<thead>
<tr>
<th>Analyze</th>
<th>The test process</th>
</tr>
</thead>
<tbody>
<tr>
<td>For the purpose of</td>
<td>improving</td>
</tr>
<tr>
<td>With respect to</td>
<td>Efficiency</td>
</tr>
<tr>
<td>From the viewpoint of</td>
<td>the test team</td>
</tr>
<tr>
<td>In the context of</td>
<td>product X</td>
</tr>
</tbody>
</table>

---

\(^{16}\) Björn Lindström, *A Software Measurement Case Study Using GQM*, 2004

2.3.3 Second step: Defining questions

The next step when using the GQM method is to derive questions from the goals. The questions have the purpose of creating a bridge between the more abstract goals and the metrics. The questions should be created in such a way that the answer to the question shows if a goal has been achieved or not. Generally many questions have to be formulated to fully cover the scope of a goal. To give an example, three questions have been formulated below. They are derived for the example in Table 2.

Q1. What is the competence level of the testers?
Q2. What test specifications are given to the external test organizations?
Q3. How much is the total cost of testing?

As seen above the questions can be of various formats and have very different purpose. Some questions will lead to more general descriptions of for example work processes while other questions exactly point out indirect or direct metrics. There are other important distinctions between questions: some lead to soft metrics and some point out hard metrics. This distinction can be made before the actual collection of metrics has begun and gives a good overview of which metrics are based facts and which are not. Another valuable tool when having defined the questions is prioritization; to prioritize the questions on importance is one way of planning and scoping the study.

2.3.4 Third step: Identify metrics

After the questions have been defined, metrics need to be identified and linked to each question. The main focus should be on what data is needed to answer each question and not on whether the data is available or not. The refinement and analysis of the collected metrics should be done after all metrics have been identified. At that point a re-organization of metrics into groups such as ‘impossible to collect’ or ‘too expensive to collect’ can be made.

M1. Total salaries of the test organization
M2. Competence of testers

The dependencies of the example questions and metrics are presented in Figure 8. For example, M2 directly answers Q1 while M1 will be used to calculate the answer to Q3. The second question, Q2, will have to be described using words or a template.

By creating goals, questions and linking them to metrics, data extraction will be made in a more structured way: each metric will have a clearly defined purpose and a traceable dependency to the defined goals. This facilitates making analyses on the collected data and helps drawing conclusions on improvement suggestions.

Figure 8. Dependencies between goal, questions and metrics.
2.3.5 Iterative method

While the GQM process can be defined in three major phases, the process itself is highly iterative. After defining the questions, for example, the goal of the study might need to be reformulated or new goals might need to be created. Hence, after defining metrics the questions might need to be re-grouped. The idea is to constantly re-work and refine all goals, questions and metrics until the best structure is reached.

The GQM method is well-suited to be used as a tool to continuously improve software processes, where the goals, questions and metrics are refined after each finished project. This way improvement or analyses can be done not only based on one, but on several projects’ experience. One extension of the GQM method that focuses on implementing several, consecutive GQM studies is the V-GQM\textsuperscript{18} method. With this approach GQM can be seen as a part of bigger measurement program, aimed at continuously improving and optimizing different areas of projects.

\textsuperscript{18} T. Olsson, P. Runesson, \textit{V-GQM: A feedback approach to Validation of a GQM study}, 2001
CHAPTER 3: METHOD

CONTENTS:

Work process
Scope
3 Method

The goal of creating a strategy that optimizes system testing at SEMC is a wide goal that can be studied from many angles. Many factors affect system testing strategies, some more important than others. The Goal/Question/Metric (GQM) method provided a way to extract the right information and to prioritize which parts of the study were most important.

3.1 Work process

The work process defined and used in this Master Thesis is shown in Figure 9. The GQM method was used more as a guideline than a rule, since there are some basic differences between generic GQM studies and the study made in this Master Thesis. GQM studies are generally used when there is a great amount of metrics and it is important to focus the data collection and collect only relevant metrics. This study was based on interviews and soft metrics, but shared the need to maintain a clear goal throughout the work process. The work process is divided into 7 basic steps all of which are described in the following chapter.

1. Conduct pre-study
2. Define GQM goal
3. Conduct focus group meetings
   Identify improvement areas & solutions
4. Create GQM plan
   Define questions & metrics
5. Collect metrics
6. Analyze results
7. Present final results

Figure 9. The work process used in this Master Thesis.

1. Conduct pre-study

Before the GQM study had begun a pre-study was made of how development and testing is performed at SEMC. This resulted in a good overview of the problems and needs of the System test organization. The results from the pre-study are presented in Chapter 4.

The pre-study generated, in combination with theoretical knowledge from the field of Software Engineering, a clear goal and definition of the Master Thesis project.
The aim of the pre-study was to generate general knowledge of SEMC’s development process as well as more detailed theoretical knowledge of product lines and testing.

2. Define GQM goal
The goal of the Master Thesis was to create a system testing strategy at SEMC, with a special focus on how the benefits of product lines could be exploited. The goal of the Master Thesis was formulated into a GQM goal. This resulted in a precise goal that was used to further analyze the problem. The GQM goal was established using the format described in 2.3.2. The final GQM goal used in this Master Thesis is presented in Table 3.

<table>
<thead>
<tr>
<th>Analyze</th>
<th>Product Line testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>for the purpose of</td>
<td>Optimization</td>
</tr>
<tr>
<td>with respect to</td>
<td>Effectiveness and efficiency</td>
</tr>
<tr>
<td>from the viewpoint of</td>
<td>System test at SEMC</td>
</tr>
</tbody>
</table>
| in the context of the | The specific family project

3. Conduct focus group meetings
A set of potential improvement areas were identified using the GQM goal, without help from SEMC employees. This was done to preserve the viewpoint of the outsider and generate thoughts that could serve as a framework for the study. After the framework was defined three focus group meetings were conducted with personnel from the System test organization.

The first focus group meeting was held together with test leaders at SEMC. A general discussion was held were the focus group was asked to identify the areas at system test where most improvement was needed. After that, a second focus group meeting was held to identify how these areas should be prioritized. This was done with emphasis on what was feasible and least costly. Solutions were also identified for each area. The GQM goal was used during these two meetings to focus the discussion on the overall goal of the study.

Finally a third focus group meeting was held together with system test management to verify that the improvement areas and solutions identified were in line with their expectations. These improvement areas were used as input to the GQM plan.

4. Create GQM plan
By using the improvement areas collected in step 3, a GQM plan was created. Data collection was structured by reformulating the improvement areas into questions and finding relevant metrics connected to each question. The purpose was to find metrics that would answer the questions and use those answers to improve the test process the way the GQM goal described. The GQM plan also provided a time plan for when the studies of the different questions should be completed.

19 The name of the project needs to be excluded, for reasons of confidentiality.
5. **Collect metrics**
By following the GQM plan relevant metrics were collected. This was done by studying literature, conducting interviews and measuring data. The identified metrics can be found in Table 4, found in Appendix A.

6. **Analyse results**
When the data was collected it was analysed with the emphasis on how it contributed to answering the questions posed and to the overall GQM goal.

7. **Present final results**
First the results from the GQM study were presented at a section meeting at system test. Then, to ensure quality and usefulness of the developed strategy, a presentation was made for SEMC employees to collect feedback. The final results were presented in a presentation and in a final report. A document containing suggestions on how SEMC can continue working with the problem of this thesis was created and handed over to SEMC at the end of the Master Thesis project.

3.2 **Scope**
The SEMC test organization is large and the number of factors that affect the test process is high. Therefore a number of areas that can be optimized using the benefits of product lines were identified and prioritized together with domain experts, as explained above. One specific product family was chosen as the main object of the study. This was done to reduce the scope and focus the study further.

The areas were identified and prioritized, mainly using focus group meetings. These areas were considered the scope of the thesis since they pinpointed the areas that need to be adjusted to support product line development. The most important areas were studied and analyzed first; the areas with lower priority were studied less or not at all. Chapter XX describes the improvement areas and how they were prioritized. Two improvement areas were considered out of scope for the Master Thesis.
CHAPTER 4:
PRE-STUDY AT SEMC

CONTENTS:
- Product family description
- Product lines at SEMC
- Configuration management
- Test process
4 Pre-study at SEMC

This chapter presents the results obtained from the pre-study made during this Master Thesis. The aim of the chapter is to explain how SEMC works today. It also explains how system testing at SEMC is performed and how the product line architecture at SEMC is structured. Configuration management at SEMC is described since it has a major role in development and testing in a product line environment. The thesis was focused on one specific product family and that product family is described.

4.1 Product family description

Four products from the same product family were the basis of the study in this Master Thesis. Their internal relations are complicated since every product consists of parts that are product specific and parts that are inherited from the mother product. Figure 10 shows how all three daughters inherit their main functionality from the mother product. The mother product is developed first and Daughter 1, Daughter 2 and Daughter 3 all follow the Mother project in time.

![Figure 10. Description of four products from the product family studied in this Master Thesis.](image)

The daughters are all based on the mother product, some components are exactly the same as in the mother, some are slightly changed and some are totally new. Different hardware and functionality requirements decide how much of the mother product that can be re-used in the daughters. The main components in a product are the application software, the signalling platform and the hardware. They all affect re-use in different ways. Figure 11 shows an example of a daughter product and its different components.

**Application Software Platform**

The application software platform constitutes all the software found in the mobile handsets. It includes all the functionality that can be used by the end-users such as the calendar, phone book and camera. Most of the application software is developed at SEMC but some parts of the system are implemented by subcontractors. The re-use of application software is controlled by a software product line which means that a majority of all the daughters’ application software is inherited from the mother product. However, most daughters also have their own product specific software.
**Signalling Platform**

The signalling platform contains functionality for the communication between the mobile handset and the hardware. The hardware needs to communicate with different networks which mean that GSM/GPRS (2G) and WCDMA (3G) products use different signalling platforms. In the example product family described in Figure 10 Daughter 2 is a 3G phone and therefore has a different Signalling platform than the other products.

In Figure 12 two different products are described to exemplify how different functionality affects re-usability. The squares within the dark dotted line describe functionality included in the daughter product while the squares within the light-coloured lines belong to the mother. As seen in the figure daughter products may contain functionality that does not exist in the mother and vice versa. In the product family studied in the Master Thesis this occurs in Daughter 2.

**Hardware**

The hardware is the physical part of the product and can be very different for products in the same family. Examples of parts of the hardware that may differ between products are: cameras, communication-ports, memories, and LCD-displays. The most obvious hardware
difference is the physical shape of the handset. Some have the standard stick shape while others can be folded like a clam, other may have two displays: the designs vary. Camera quality and memory types are differences that are not obvious but that may affect re-usability greatly.

4.2 Product lines at SEMC

The SEMC equivalence to a product family is called the application software platform and it is towards that platform developers work; implementing new code or applying changes to existing software. Individual products are created from the main platform by specifying which functionality should be included in the product. When errors are found during testing, they are reported on the product in which the error was found, but the corrections are made on the main platform. The individual products are thereafter re-built based on the main platform’s changes. The main platform and how it relates to different products is described in Figure 13.

![Figure 13. Description of the main platform and how it is used in development and testing.](image)

4.3 Configuration management at SEMC

Developing software for mobile handsets is a complicated process that requires a high degree of planning and synchronization. Since the software in a mobile handset is very large (several million lines of code) a large number of developers are needed. Using product lines to save resources and increase the size of the product portfolio adds to the need of control and code maintenance. This puts high demands on Configuration Management (CM).

As explained in the Chapter 2.2.3 CM keeps track of changes of the code and builds executable software out of the developers’ code. Since developers at SEMC work towards the application software platform, without focus on specific products, CM has to configure each build to suit specific product needs. Configuration is done by setting variables that together define the selected product. These configuration variables are used to select which parts of the software that should be included in a build. The configuration variables typically describe how the software should communicate with the hardware. They contain information on display size, FM radio, Bluetooth etc. However, there can be several variants of each product in the product family, e.g. a product released in the Chinese market needs to be adopted to
meet the needs of that market. CM must therefore be able to build all different products and each product's different variants.

Since mobile handset software tends to be very complex and there are many different variants that need to be built, the number of configuration variables is high, usually several thousands. This makes it hard to keep track of the exact difference between products. In a fast-paced environment like mobile handset development changes are implemented quickly. Builds are made very often, in the last phases of development builds are made every day.

Figure 14 shows the interaction between the main platform, developers, CM, and software builds. It shows how the code is created, updated and corrected by the developers and thereafter built by CM. Testing of the products is performed both on the platform and on the products: different kinds of tests depending on the test object.

### 4.4 Test process

Figure 15 shows an overview of how code is assembled to a complete product and how testing is performed. CM’s role is also evident; they assemble all available functions and build the software for each product. A special CM responsible in the function test organization assembles components into functions.
The first type of testing, *unit testing* is performed by the developers. They test their own pieces of code, called units. Several units are then combined into a component which is *component tested* by the developers and verified by function testers. After that the Function Test group performs tests on several components. The developers and function testers are grouped based on functionality, so *function testing* is performed with a focus on single functions. Further testing is performed by the System Test group, a separate test organization that focuses on verifying and validating the overall functionality of the product. *System testing* is described in more detail in chapter 4.4.1. Finally a *Customer Acceptance Test* (CAT) is performed in which customers ensure that the final product meets its requirements.

The test process used is highly iterative which means that errors found during the different tests generate new code pieces (or corrected code) and thereafter the test process restarts. The different types of testing are performed more or less at the same time. Component and function testing is performed in close cooperation while system testing is performed separately. The majority of the customer acceptance testing is performed separately from the other test types, by staff assigned by the customer.

### 4.4.1 System testing

System testing at SEMC is performed by a specific system test organization that focuses on testing the complete product. Interaction and regression tests are performed and several other types of testing are controlled from within the system test organization. Field tests, CAT and release tests are examples of tests that are supervised by the system test organization.

Figure 16 shows two important dates in a products development timeline is; *alpha* and *launch date*. Alpha is set to when the test object has enough functionality and the product is considered to be complete enough to be system tested. Ideally, no new functionality should be implemented after Alpha has been reached. Alpha is when system test begins. At a later stage
of the development, when the product has been tested sufficiently, a decision has to be made whether to release the product on the market or not i.e. does the product meet the requirements of the market? If the answer is affirmative the product is released. That answer affects the launch date which can significantly decide a product’s success on the market. Testing after the launch date is called maintenance, since it involves maintaining and improving the already existing, released product. As described in Chapter 2.2, the maintenance phase eventually evolves into a new project which uses the old product as a starting point.

Regression testing at the system test level is performed continuously to ensure that important software builds still are of high quality. The continuous regression testing is specified in the test plan. However, there is sometimes a need to make unplanned regression tests, for example when big changes have been implemented. These regression tests need to be executed fast and cover a large quantity of the functionality. These regression tests are referred to as release tests. Release tests are constructed so that they exercise all the parts of the mobile handset superficially to expose areas where new errors have been inserted. Running a release tests is a quick way to get an overview of where errors have been introduced. During system testing at SEMC most of the testing is performed to find regression errors.

In general, all kinds of system testing are based on executing test cases from the test specification. Test specifications are developed based on the requirements specifications, to ensure that the tests verify system quality. When creating test specifications for daughter products the mother products’ specifications can be used, transforming it to also include daughter specific requirements. This way test specifications can be re-used and don’t need to be re-developed for every new project. Test specifications are re-used between consecutive product families as well as between the products in a product family.

Since testing sometimes is repetitive and mechanical, some parts can be automated. By letting computer programs execute test cases time and valuable resources can be saved. The automation at SEMC is focused on automating larger parts of the system test suites. With a fully functional test automation system testing can be performed faster and cheaper.
CHAPTER 5: RESULTS

CONTENTS:
- Software Variability
- Hardware Variability
- File system Variability
- Signalling platform Variability
- Marketing Variability
- Configuration Variability
- Test case selection
- Specifications
- Product family planning
- External relations
- Operator Variability
- 6th sense decisions
5 Results

This chapter presents the results gathered from the GQM study made during the Master Thesis project. The work process in Chapter 3.1 describes how the results were obtained. The problem of how product line benefits can be used more efficiently at SEMC was analysed from several perspectives. Dividing the problem into improvement areas provided a good way to structure and prioritize the study. The final prioritization of these improvement areas was made by combining the results from focus group meetings conducted at SEMC with theoretical knowledge from the field of Software Engineering. Figure 17 shows the improvement areas, ordered by importance; it also provides an overview of the outline of this chapter. Sixth sense decisions and operator variability are improvement areas that were considered to be out of scope for the Master Thesis.

![Figure 17. The improvement areas studied in this Master Thesis.](image)

The most important issue when optimizing system testing, using the benefits of product lines, was to establish the differences between the products in the product family and also to some extent between different product families. This is referred to as establishing the product variability. Product variability can be extracted on three different levels; variability between product families, variability between products, and variability between different product builds. Figure 18 shows the relationships between the different types of variability.
Defining the product variability is essential when optimizing system testing because it allows the testers to avoid duplicate testing. Product variability can also be used to monitor how well the product portfolios follow the product line strategy i.e. it is a control measure for the variability that exist between product families, products in a product family, and product builds.

Figure 19 shows the product variability of the mobile handsets developed at SEMC divided into six areas: software, hardware, file system, signalling platform, marketing, and configuration variability. It was necessary to establish these differences since the product line architecture used at SEMC did not provide information about exactly how the products where configured i.e. it did not provide a measure of the product variability in and between the product families.
The importance of monitoring and keeping track of product variability has been documented in several research articles\textsuperscript{20}. It was also affirmed by the focus group meetings conducted in this Master Thesis project. When working with product lines it is optimal to keep the product variability to a minimum from a development and test cost perspective. Chapter 5.1 to Chapter 5.6 describes the different areas of product variability established for mobile handsets developed at SEMC.

5.1 **Product variability - Software**

**Underlying question:** What are the software differences between products in a product family?

The software variability was established as the most important variability to extract at the focus group meetings held at SEMC. The software variability gives the system testers information about where to focus their testing, both with regard to products in the product family and to the releases of new builds. The ability to establish a test object that mirrors the variability between different builds significantly reduces the time spent on system and regression testing. This fact becomes evident when comparing the amount of tests run in a complete regression test suite, to the amount of time spent testing the variability during a regression test suite. This is further described in Chapter 2.2.2.

5.1.1 **Current Practice at SEMC**

At SEMC the software variability resides only in the requirements and system specifications. This is a high level representation of the variability that is not easily transferred to system testing since SEMC does not maintain a one-to-one correlation between requirements and test cases. Furthermore the configuration of the different products is made by setting configuration variables during compilation. To obtain the software variability at SEMC the code itself has to be analyzed. At the starting point of the Master Thesis project SEMC did not extract the software variability between their products.

5.1.2 **Implementation**

Having established that the software variability could only be obtained by looking at the variability of the executable code, a tool that could extract the software variability was built as a part of the Master Thesis.

To extract the software variability between products, function length was chosen as suitable abstraction level. By comparing function\textsuperscript{21} lengths between two software builds a quick estimation of whether the code had changed could be made. A function could have been changed, added, or removed. The reason for choosing function length as a suitable abstraction level was that the platform is not configured into individual products until compilation, when the product configuration variables are set. Therefore it is not possible to extract software


\textsuperscript{21} This function refers to a C-code function and should not be confused with the function mentioned earlier which is a grouping of functionality.
variability information before the product is compiled. When the products are configured and compiled, assembler code is generated. A file is also generated that contains information about assembler function start- and end-addresses in the memory, and which file the function resides in. This file could easily be used to establish whether function length had changed between two builds.

The software variability is defined as the test object. The test object includes changed, added, and removed functions. The information about function lengths was used to populate a database with data from builds from relevant product families. The software variability could then be extracted by doing database queries. A tool was built to facilitate the database searches. The tool defined the software variability on three levels: changed functions, added functions, and removed functions, as seen in Figure 20.

Figure 21 further describes how the function length changes in a new product build are detected when the comparison between the two builds are made. The functions that are identified as changed are collected in the test object, together with the added and removed functions. In Figure 21 this means that Function 4 in Build 1 needs to be retested in the new build. Since no functions differ between Build 3 and 4 and no testing needs to be done on Build 4.
The tool was used to extract data about the variability between products in the product family, between builds of different products in the product family, and finally between product families. The test object is referred to as the list of functions that need to be re-tested. The *software delta* can be used as a measure of the software variability it is defined as the sum of all percentages, as seen in Figure 22.

![Software delta formula](image)

**Figure 22.** The definition of software delta used in this Master Thesis.

Figure 23 shows the deltas between the major builds such as the Launch build, the first Maintenance build, and the second Maintenance build, of the Mother, Daughter 1 and Daughter 2 in the product family studied. It shows that large changes were introduced during the maintenance phase of the Mother while the daughter products’ software remained more stable. However between the launch and the first maintenance build the Mother was more stable than the daughter products.

![Products build Variability](image)

**Figure 23.** The delta between builds of products in the same product family.

---

22 All numbers have been removed because of security reasons.

23 Daughter 3 was excluded because of insufficient data.
Figure 24. The delta between products in the same product family.

Figure 24 shows the delta between the Mother, Daughter 1 and Daughter 2 in the product family studied. When looking at variability in a product family the mother product is used as a point of reference and the daughters in the product family are compared to that product. Figure 24 shows how the software delta between Daughter 1 and the Mother was smaller than the software delta between Daughter 2 and the mother product.

### 5.1.3 Analysis

The analysis of the software variability shows that it can be monitored on three levels. Software variability exists between different builds of the same product, different products in a product family and between different product families. The analysis has also shown that there are several benefits to monitoring the software variability. Information about software variability can be used in product portfolio planning, to establish a regression test object, and to monitor function growth during development.

**Monitoring and planning the product portfolios**

The information about software variability between product families can be used to monitor the product line strategy. The size of the delta between product families is a key performance indicator (KPI) that can be used to evaluate the product line strategy used at SEMC. If the delta is very large between different product families this indicates that the product families should be developed with a greater emphasis on re-use of software from previous product families. The software variability is an important measure of the success of the product line strategy used at SEMC. For example, Figure 24 shows how Daughter 1 shares more functionality with the mother product than Daughter 2. This could mean that the product family has been planned and developed without enough emphasis on re-use.

The software variability can also be used to find hidden costs in development. If the products developed for a product family are very diverse i.e. have a high level of variability, this is an indicator of that the code is not developed with a strict product line focus. When a strict product line focus is maintained, modules and functions that perform similar activities are re-

---

24 All numbers needs to be excluded, for reasons of confidentiality.
used in all the products in the product line. If a strict product line focus is not maintained the variability between the products in the product family grows.

![Figure 25. Examples of high and low re-use.](image)

The cost of this unnecessary development is not seen until a delta between products in a product family is calculated; these are so called hidden costs. The delta is therefore also a measure of how well the development organization has absorbed the product line strategy. Figure 25 shows two examples of how re-use between two products can be made; the first is the optimal case when most of the functionality is re-used; the second is an example of low re-use, when the same functionality is implemented several times. The uneven area in the second image represents functionality that already exists in the mother product but is implemented once more for the daughter. Monitoring software variability can minimize this problem.

**Establishing a regression test object**

The software variability can be used to focus the regression testing by creating a smaller test object. The functions that differ between two builds can be used as a starting point for the regression testing. The test object pinpoints the functions that have been changed between two builds. These are the functions where the regression testing should be focused since this is where erroneous code may have been introduced since the last build.

![Figure 26. Establishing a regression test object](image)

Figure 26 explains how the regression test object may be reduced using the software variability. The test object is minimized by removing functions that exist in both the first and
second build that have not changed. The functions that are not changed between two builds do not need to be retested in the new build of the mother.

**Establishing a daughter test object**

The software variability can be used to further reduce the test object when regression testing daughters in a product family. The daughters are essentially built using the same software as the mother products. They are distinguished by their difference to the mother product. When a new build of the product family is made only the difference to the mother needs to be regression tested on the daughter. The functionality that exists in both the mother and the daughter should only be tested on the mother to improve efficiency. When a product has reached Alpha state no new functions or functionality should be added. Therefore the daughter test object does not change if the product has reached Alpha state. This fact can be used to extract an even smaller test object for the daughter by defining the variability between the daughter test objects of different product builds.

![Figure 27. Focusing the regression testing of the daughters in the product family.](image)

Figure 27 explains how the daughter test object variability can be extracted. Functions that have not changed since the previous build of the daughter do not need to be retested in the new build thereby reducing the test object further.

**Monitoring function growth during the product life cycle**

The software variability established between product builds can also be used to monitor the function growth during the product life cycle. Ideally the function growth should be zero when a product has passed Alpha state. By monitoring the software variability between different software builds unwanted function growth can be detected. To monitor function growth is important when determining how many resources are needed in a product project. If the function growth is unpredictable more resources have to be allocated to compensate for this fact. Being able to monitor function growth enables the test organization to plan their resources in a more structured way.
5.2 Product variability - Hardware

**Underlying question:** What are the hardware differences between products in a product family?

Hardware plays an important role in embedded systems and affects system testing a great deal. Most faults are found in the software many problems occur when the software and hardware are integrated. The interaction between hardware and software can produce errors that are both hard to find and costly to correct. For example, different sizes of displays or different type of memories can lead to complicated errors. It is therefore vital to analyze hardware differences between products (hardware variability) in a product line. By knowing where the products differ, potential problems can be avoided or at least minimized.

It is important to note that hardware is not developed using a product line strategy. On the contrary the hardware is used to differentiate products with a very similar software configuration. When it comes to hardware, re-use can be made between components that are the same in several products, but not based on a product line.

### 5.2.1 Current practice at SEMC

At SEMC today hardware variability in product families is taken into account by the test leaders in each project when they select what test suits to run. They make those decisions based on several facts, among them hardware variability. At present time, the hardware variability is maintained in the requirements documentation. Hardware variability between different products in a product family is extracted for each product. Detailed information about hardware specifications are maintained by the hardware projects themselves. An estimation of the hardware differences between the products in the product family studied in this Master Thesis show that up to 50% of the hardware differ between the mother and the daughter product that are most different. The daughter that is most similar has identical hardware configuration. The two daughters that were most different were partly developed at another site and that fact together with the big differences in hardware forced testers to treat them as separate products.

### 5.2.2 Suggestion for implementation

By collecting the hardware variability and monitoring changes to the existing hardware components decisions about where to focus the testing can be made easier for the test leaders. Statistics of which hardware changes have introduced the most faults can help improve these decisions by pin-pointing where hardware will affect system testing the most. With a clear focus on hardware variability and its interaction with the software, product line testing can be made safer and more efficient.

### 5.2.3 Analysis

The hardware variability exists on a component level. However the hardware is not developed using a product family strategy and therefore hardware variability of a product family can not be monitored using the product family concept. Instead hardware variability must be

---

25 The percentage has been calculated by making a list of hardware components for the mother and for the daughters and thereafter calculating a percentage of how many components differ from the mother.
monitored on a component level. Hardware errors that are found in one component can be expected to be found on similar components.

5.3 **Product variability - File system**

**Underlying question:** What are the file system differences between products in a product family?

The file system is the part of the software that is not executed during run-time. It consists of different components that are vital to the system. Examples of such components are device drivers, menu trees, pictures and language packages. The difference between file systems is of great importance when performing system tests. For example, a change of driver for the camera affects all camera related test cases and the test focus should in this case be changed towards camera functionality. However, not all changes in the file system will have high impact on the testing. Moreover important file system changes do not occur as often as software changes.

5.3.1 **Current status at SEMC**

Today the file systems are updated and maintained by the different development groups individually. This result in that information about file system changes not being communicated to the System Test organization. The test leaders are therefore not able to take file system changes into account when deciding how to focus the testing. Important file system changes have been communicated on an ad-hoc basis i.e. when needed.

5.3.2 **Suggestions for implementation**

A complete and good way to monitoring the file system changes is to save information on every new file system into a database. This way, a comparison between file systems can be made the same way comparisons are made between different software builds, as explained in Chapter 5.1. The file system variability can be completely monitored by keeping track of changes on hardware drivers, graphical objects and menu trees.

5.3.3 **Analysis**

The file system variability needs to be monitored for each product the same way the software variability is monitored. Every change in the file system has to be analyzed and the effects of that change need to be identified in order to know where to focus the testing. The file system variability has to be monitored on a file level; every update of a file in the file system is extracted as file system variability.

5.4 **Product variability - Signalling platform**

**Underlying question:** What are the signalling platform differences between products in a product family?

The signalling platform in the mobile handsets controls the communication between the mobile handset and the network. The hardware needs to communicate with different networks which mean that GSM/GPRS (2G) and WCDMA (3G) products use different signalling platforms. The structure of a typical mobile handset is further described in Chapter 4.1.
5.4.1 Current practice at SEMC
The signalling platform developed at SEMC is bought from a subcontractor. The subcontractor develops platforms for a number of different customers of which SEMC is one. The software of the signalling platform is delivered as a module and is integrated into the software application platform when the products are configured. Platform tests are run every time a new platform is developed or a new release of the platform is made.

5.4.2 Suggestions for implementation
The signalling platform variability can be established using the same method as was used to extract the software variability. Special naming conventions are used to name functions that are part of the signalling platforms. The measurement of the signalling platform variability is important to monitor since the change logs delivered from the subcontractor might be unreliable. Their platforms are developed and sold to several different customers and the platform delivered to SEMC may include corrections and adjustments required by other customers than SEMC. This has led to that new signalling platform builds from the subcontractor have to be monitored and tested thoroughly to assure that they do not introduce unexpected faults in the application software platform.

5.4.3 Analysis
The signalling platform variability exists on two levels. The signalling platform differs between different software builds, since corrections are inserted continuously. There is also variability on a larger level, when new versions of the platform are created. The difference between these versions can be found when investigating the product family variability, since new versions of the signalling platform are inserted in new product families.

The possibility to monitor changes in the signalling platform has several advantages. First of all it enables SEMC to follow up the requirements on the code delivered from the subcontractor. It also enables SEMC to make decisions on whether or not to take in a new build of the platform, since the changes show the potential impact of including the new signalling platform build in the product. Finally it enables SEMC to focus their testing on the actual changes of the product, including changes of the signalling platform.

The internal structure of the subcontractors signalling platforms affect the structure of SEMC’s application software platform. The signalling platform is the basis of the mobile handset; it delivers the basic phone functionality. Therefore the development of the signalling platforms greatly affects the development of products at SEMC. A non-modular structure of the signalling platform has led to a non-modular structure of the application software platform. If the mobile handsets developed at SEMC were built using an operative system SEMC would not be equally dependant of the code delivered from the subcontractor. With a well-functioning operative system the interaction between hardware and software could also be simplified, since operative systems usually handle communication between software and hardware.

5.5 Product variability - Marketing

Underlying question: How are the marketing differences taken into account when testing a product family?
The concept of product lines is especially efficient when having small differences between products since it reduces development costs. However, it is important to market products as being unique: to have different functionality focus on different products in the same product family. For example, one product can be designed for children while another is developed to meet the needs of construction workers. These two products would have different end-users with very different demands on functionality and therefore also different marketing focus. The difference of marketing focus needs to be taken into account when testing large product families with many different products. On a phone marketed towards construction workers durability and drop tests would be very important while products developed for children need to focus on testing games and entertainment.

5.5.1 Current practice at SEMC
The difference in marketing focus is not explicitly taken into account during system test today.

5.5.2 Suggestions for implementation
Prioritizing of test cases can be done depending on the marketing focus which establishes the product’s intended end-user. Using statistical information about which functionality the intended end-users use more creates an image of the end-user and his or hers needs. The functions that are used more often should be tested more thoroughly and functions that are used seldom or not at all can be tested only when time and resources allows. This provides a way to get an individual test prioritization based on the marketing focus for each product.

This is often refereed to as usage-based testing\(^2\). Statistics on usage can be used to prioritize through the whole development process. When products need to be developed fast usage statistics help prioritize which functionality a product should contain. This enables products to be developed and tested based on the needs of the market.

5.5.3 Analysis
The number of daughters in today’s product families is fairly limited and a clear marketing focus on products has not been necessary; prioritization has been done based on other facts. However, in a product family with more than ten daughters it will be vital to know how each product should be tested and what functionality is most important to the product’s end-user.

5.6 Product variability - Configuration

**Underlying question:** Can the configuration variability be specified on an abstract level?

Configuration variables are used to differentiate products in a product line. A configuration variable is a mechanism in the configuration file that enables or disables functionality in the software.

--
5.6.1 Current practice at SEMC
At SEMC the software application platform is the basis of all product families. The individual products in the product families are not configured until they reach the compilation stage. At compilation all the configuration variables are set. The configuration variables are further explained in Chapter 2.2.3. Each function group maintains the configurations variables that are used in the function that they develop. The configuration variables define the behaviour that differ between products in the product family i.e. they define the variability between products.

There is no common documentation and definition of the configuration variables. It is configuration management’s task to collect the information about what configuration variables should be set in each function and product.

5.6.2 Suggestions for implementation
The number of configuration variables in a product family is very large. A change of one configuration variable may alter the behaviour of all other configuration variables. This complicates both the development processes and the configuration process.

Information about which configuration variables should be set for a product need to be declared and described by the developers. This information could then be extracted and used in the engineering process to increase control of configuration variability.

5.6.3 Analysis
The study of configuration management shows that the configuration variables should be specified on a higher abstraction level than the one used today. How this problem has been studied in more detail in another master thesis conducted at SEMC\(^2\)\(^7\). Another improvement would be to have better encapsulation because this would facilitate the build process and reduce the number of configuration variables. To investigate how the encapsulation could be increased was seen as outside of the scope of the Master Thesis project.

5.7 Test case selection

**Underlying question:** Which test cases should be run on the mother product and on the daughter products?

The way test cases are chosen affects many parts of testing and development; profitability, efficiency and effectiveness are all directly dependant on if testing is focused the right way. However, having a clearly defined test object is something that is required before test case selection can be performed.

The most fundamental problem when trying to optimize testing is how test cases should be selected. That question addresses the underlying issue of how testing should be focused i.e. which parts of the products should be tested more thoroughly. In general, test cases are

\(^2\) Björn Pileryd & Anders Hellström, *Controlling the variant explosion*, 2005
written and collected in the test specification in the beginning of a project. These test cases are then used as a set of possible tests, preferably covering all different parts of the test object. This set of test cases is usually referred to as the test suite. When optimizing testing the most fundamental question that has to be answered is which of the test cases from the test suite should be executed.

Since testing often is done in many small increments the decisions on how to choose test cases are made often and fast; sometimes based solely on experience and feeling without any hard facts to back it up. Choosing test cases has to be done at different times; after a software build has been finished or when a daughter product is to be tested. In both these cases extracting a product differences is the way to decide where to focus testing. To choose test cases based on implemented changes is referred to as change-based test selection.

In Chapter 2.2.2 one method to ensure good regression test selection is described, showing how new software builds can be tested in an efficient way. This method can be applied at SEMC and used to enhance regression test selection on all products.

The same test case selection method can be used when choosing which test cases to run on daughter products, when a large amount of testing already has been conducted on the mother product. The mother product can be seen as an earlier version of the daughter product which enables the use of regression test techniques on daughter testing. By extracting the variability between the mother and the daughter product a test object is established that points out where to focus the testing on the daughter. The daughters should be focused on testing the variability since the functionality the mother and daughter have in common has already been tested in the mother.

The change-based test selection technique can be divided into three steps:

**Step 1: Compute software difference**

As described in Chapter 5.1 the software variability can be established between two software builds. That variability can be used as a first step in this test selection method. By knowing which functions are included in the test object, test cases can be chosen in an efficient way. Figure 28 shows an example of the variability between two builds.

**Figure 28. Difference between two software builds.**
Step 2: Coverage Analysis

By making a coverage analysis of which code is executed by which test case the results from Figure 29 can be obtained. This can be performed by run-time collection of coverage data during the execution of test cases. Figure 29 shows how this information creates a mapping between test cases and executed code. When Test Case 1 (TC1) was performed functions 1, 2, 3 and 4 were executed.

Figure 29. Mapping test cases to executed functions

An approach to reduce the analysis time and the size of the regression test suite is to re-test only the test cases executing changed and removed functions. Since the mapping of test cases (step 2) is performed using the code from the first build, added functions will never be connected to any test cases. The added functions have to be taken into account by analyzing their impact, i.e. investigating how their new functionality affects the test object.

In Figure 28 Software build 1 is the build from a product X and build 2 is the new build for that same product. When the new build is to be tested there is a need to know which test cases should be run. The difference in Figure 28 shows that Function 5 has been changed and Figure 29 shows that function 5 is exercised by Test Case 2. The resulting test selection using change-based test selection would be to only perform Test Case 2 and not Test Case 1, since Test Case 1 does not exercise changed code.

The example from Figure 28 and Figure 29 is a theoretical example used to show how change-based test selection is used. However there is one potential threat to this test selection method: if the number of functions exercised by all test cases is large the method will select all test cases to be rerun every time one of these functions is changed. Examples of such functions could be functions that control graphical operations, start-up functions or functions that are executed periodically during execution. There is a way to avoid this problem though, and by doing that gain other advantages as well. A suggestion on how to resolve this problem is suggested in 5.8.2.

5.7.1 Suggestions for implementation

In order to implement and start using change-based test selection at SEMC some practical problems have to be solved. These problems were identified and analyzed during a feasibility study of how a change-based test selection method can be applied at SEMC. The first step of the implementation was to find a way to extract software differences on products. This step is described in detail in Chapter 5.1, since it itself proved an important part in optimizing system testing. The second step, the coverage analysis, proved to be somewhat more complicated, mainly due to the enormous size of code to be covered (millions of lines of code). The feasibility study’s main focus was on how the coverage analysis could be performed and the detailed results of the study is presented in a separate document. A summary of these findings is presented in the following text.

Two different main ideas to perform a coverage analysis were explored during the feasibility study: a method using hardware analyzers to extract runtime execution information from the
processor and a method using *profiling* to change the executed code and log every time a function was executed. The first method is hardware-based and the second software-based. These two main ideas proved to have their own difficulties, limits and benefits.

**Hardware-based code coverage**

Hardware-based code coverage is based on having a hardware tool that can monitor the activity of the processor on which the code is executed. Different tools can be used to accomplish this, each with their own limitations and benefits. Hardware tools are usually expensive and require expert knowledge to operate. However, the hardware-based coverage has the benefits of not interfering with run-time execution. Since many of the important functions in mobile handsets depend on real-time execution interfering with it might cause the products to behave in an unnatural way. Hardware-based code coverage could prove to be a good way to avoid this interference, however during the Master Thesis project software-based code coverage was established as a better alternative. The hardware tools found at SEMC were both too expensive and too complicated to use for this purpose. Other tools might prove to work but only tools found at SEMC were investigated.

**Software-based code coverage**

The software-based code coverage explored during the feasibility study is called *profiling*. To profile code means to change it so that it during execution not only executes normally but also performs other operations such as collect coverage information i.e. log which functions have been executed. Profiling interferes with real-time execution, but if the code that is inserted is not too lengthy the interference can be acceptable. The feasibility study showed that profiling the code is a good way of collecting coverage information. A previous Master Thesis has shown that profiling is a possible way to collect information about coverage.\(^\text{28}\)

In order to choose which coverage analysis method is the better it is vital to have a clear picture of the real-time requirements. Factors, such as tool learning time and tool costs, also need to be taken into account when choosing method. After having identified these requirements and costs, it is possible to choose the best way of performing a coverage analysis. One major decision that affects the requirements greatly is how the test selection method should be used i.e. under which circumstances the coverage analysis should be performed. Performing coverage analysis during automated tests generates different demands than during manual system testing. The circumstances also decide how often the coverage information needs to be collected; which is important when calculating for example costs for tool usage. Exact figures are needed of how much real-time execution can be altered to allow realistic testing, since those figures provide a way to directly know whether a coverage method is feasible or not.

The feasibility study shows the need to approach the problem of performing coverage analysis in a well-structured manner: first of all making a complete requirement analysis and based on that choose the best implementation of code coverage.

---

\(^{28}\) Rickard Möller, *Optimizing Demand Paging using Profile Guided Function Positioning*, 2005
5.7.2 Analysis
Change-based test selection is a good way to improve test selection. It provides a safe way of choosing which test cases to execute on daughter products as well as for regression testing on all products. The previous chapter showed how coverage analysis can be used to map test cases to executed functions. However, there is different way to perform change-based test selection that does not require the use of a coverage analysis. That way has been adapted by the test leaders at SEMC.

After extracting the variability an investigation of how the functions affect the test object is made. The test cases are then chosen based on that information. For example, if all the changed functions can be connected to camera-related code, only camera-related test cases need to be executed. This requires that there is time to investigate how the changed functions are used. This information can be hard or impossible to obtain if the variability is large. It also requires that the test leaders are very familiar with the code. However when product software is re-built often, the number of changed functions is small enough to enable this kind of change-based testing.

However, using a coverage analysis to map test case to code provides a more exact way of focusing testing, since it also enables change-based testing when the variability is high i.e. when the number of functions in the test object is large. As described in Chapter 5.8.2 a coverage analysis has other benefits as well, for example when improving test specification quality. Since the profiling was not fully implemented during the Master Thesis it was not possible to investigate the costs and benefits of performing a coverage analysis at SEMC.

5.8 Specifications

**Underlying question:** How can the test specifications be created so that they mirror the difference that exist in a product family?

There are two types of specifications relevant to this thesis; the requirement specification and the test specification. They are closely related since the test specifications should reflect the requirement specifications for each product. This is further described in Chapter 2.2.

5.8.1 Current Practice at SEMC

**Requirement specifications**
The number of requirements for the products at SEMC is high, for an average project over 10000. Complete requirements specifications are typically only created for the mother project and delta requirement specifications are developed for the daughter projects.

![Figure 30. Mother and daughter requirements](image-url)
Delta specifications define the requirements that differ from the mother product. Figure 30 shows the relationship between mother and daughter requirements, and how a large part of the requirements are equal. The requirements that are specific to the daughter are collected in delta requirements specifications. Another complicating fact is that as requirements change during the course of a product's development, the requirements specification needs to be updated. The requirement specifications are owned by the development groups. The large numbers of requirements make the communication of changed requirements more difficult. Another complicating fact is that the people who update the requirements are different from the ones that monitor them. This fact stresses the importance of structuring the requirements in a way that makes them easy to maintain.

**Test Specifications**

When the test specification is developed for the mother product, it follows the requirement specification closely. When the test specifications are developed for the daughters, they are created as almost exact replicas of the mother test specification, only modified to match the delta requirement specifications. This causes a lot of unnecessary work since common functionality is tested on all the products in the product family, even though common functionality ideally, only should be tested on the mother.

### 5.8.2 Suggestions for implementation

In the mother product, all the functionality should be covered by the test specifications. Therefore, the mother test specifications should verify and validate the requirements specifications. In the daughters, only the functions that do not exist in the mother project and their interface should be covered by the daughter test specifications.

Coverage analysis is a method used to keep track of which functions are executed during a test suite. This is further described in Chapter 2.2.2. In Chapter 5.7, a mapping of which functions are exercised by which test cases is used to choose what tests to run during regression testing. This information can however, if used with care, also be used to create new test cases, create test specifications, and to group and prioritize test specifications.

#### Create new test cases

The coverage analysis shows how many percentage of the total code that the test cases, in the test suite, exercise. Traditionally in testing, this information has been used to create more test cases so that a higher level of test coverage is obtained. However, this is a dangerous path to take since it leads you to believe that the better test coverage you have, the better test strategy you are using. This is further described in Chapter 2.2.2. Instead of an analysis of the functions that are not covered by the test cases being done and if needed new test cases being developed, information about functions that are never exercised should be fed back to the development organisation to enable them to review the code.

#### Create test specifications

The coverage information can also be used to facilitate the creation of test specifications for the daughter products. Instead of creating new test specifications for the daughters based on the delta requirements, the coverage information can be used to pinpoint what parts of the mother test specification can be used as input for the daughter test specifications.
The mapping of the mother test cases to functions they exercise in the daughter will in combination with information about what functions only exist in the daughter test object, show which of the mother test cases should be run on the daughter. For the functions that exist in the daughter test object but are not exercised by any of the test cases in the mother test specification, new test cases will have to be created.

Figure 31. Creation of daughter test specifications

Figure 31 shows how the daughter test specifications can be created using the mother test specifications as a basis. The daughter test object is extracted as described in 5.1, collecting the differences between the mother and the daughter product. By performing a coverage analysis of the daughter software using the mother test cases some of the added functions can be mapped to the old test cases. New test cases will have to be created for the rest of the added functions in the test objects. In Figure 31 Function 3 and Function 5 are new functions that are tested by the mother’s test cases. New test cases only need to be created for Function 4 and Function 7.

This way the one-to-one correlation between the requirements specification and the test specifications do not have to be maintained and instead a one-to-many correlation can be maintained between test cases and the functions that they exercise. Since the functions are developed based on the requirements specifications this creates an indirect connection between the requirements specifications, the functions, and the test specifications.

Figure 32. The correlation between requirements, functions, and test cases.
**Grouping and prioritization of test cases**

The coverage information can also be used to analyse which functions the test specification exercises the most. This information can be used to prioritise test cases in the test suite. The functions that are exercised the most are probably the functions that are crucial to the stability of the product. These test cases should therefore be prioritized in a situation when little time for regression testing exists.

![Diagram](image)

**Figure 33. Prioritization of test cases based on how much the function are used in the test cases.**

Figure 33 shows how the coverage information can be used to prioritize test cases. This information can also be used to group the test cases into system-controlled and input based test cases. System-controlled test cases would then be test cases that execute functions that are invoked by real time responses such as for example network searches. Input based test cases would be test cases that are invoked by user input, for example setting up a call. Functions that have a high usage percentage are prone to be real-time functions. These functions should ideally be extracted from the user input test cases, and be tested separately as performance test cases.

### 5.8.3 Analysis

The test specifications can, by using information about the test object and the coverage analysis, be created so that they mirror the differences that exist in the product family. By using this method the cost of creating test specifications for the daughters can be reduced. This method can further be used to create test specifications for new product families. The original test specification can be re-used for mother products in following product families. Figure 34 shows how test specifications can be re-used for daughters and different product families. The arrows specify between which products re-use of test specifications is possible.
5.9 Product family planning

**Underlying GQM question:** How can the planning of what products to include in a product family be improved?

The planning of product families in the right way is probably the single action that most influence how well the benefits of product lines can be used. In general, low differences between the products lead to increased development savings but on the other hand, products need to differ in order to appeal to end-users in different customer segments.

5.9.1 Current practice at SEMC

The product family studied in this Master Thesis was one of the first product line families at SEMC. Therefore the planning of the product family did not have a complete product line focus; some early mistakes were made. The products were spread out geographically and some daughters were very different from the mother product, decreasing development and test savings.

5.9.2 Suggestions for implementation

Using statistics on product differences from earlier product families will provide a way to evaluate and thereafter enhance product family planning. Investigating the variability between consecutive product families as well as between products within the families is another way to get a clear picture of how well a family was planned. Figure 35 shows the two levels of variability that can be used to improve product planning. These differences can be obtained from comparing software builds as explained in Chapter 5.1.
5.9.3 Analysis
The future at SEMC holds larger product families with more daughters. In those projects product family planning will influence profitability greatly. It will be important not only to develop and design good products but also to group them into families in a good way so that the product line concept can be used at full advantage. By investigating the variability product family planning can be evaluated, and evaluation is the first step towards improvement.

5.10 External development and testing

**Underlying GQM question:** How are decisions made about where to locate daughter projects?

The benefits of product lines can be decreased if parts of the development are geographically spread out\(^{29}\). At SEMC not only the company’s organization is spread out but some parts of development and testing is outsourced. When using product lines it is important how you choose which parts should be outsourced and which part should not, since saving resources generally depend on good communication and re-use of knowledge and software.

5.10.1 Current practice at SEMC

Decisions regarding outsourcing are mainly made based on financial reasons; short-term testing can be performed by other companies without increasing the fixed costs at SEMC. When interacting with organizations outside the main site a greater focus needs to be put on planning the testing since cross-organizational interaction generally is needed to allocate resources in time.

One of the biggest problems regarding external development and testing, is when a daughter product is developed at a different site than the mother product. This leads to several complications and often a decrease in the possibilities of re-using software and knowledge. At SEMC today about 50% of the daughter products have some part of the software process based outside the mother product’s site. Often more than one CM organization needs to be involved when developing software at different geographical locations. CM is an important interface between developers and testers and involving several CM organizations in the same project generally leads to more work and unwanted variability between software builds. For example, several different builds are often created for the same daughters, one build at the mother product’s site and one at the site where the daughter is developed or tested.

5.10.2 Suggestions for implementation

The best solution regarding both outsourcing of tests and the placing of daughter projects at different sites is to focus the testing at one place as much as possible and identify which areas of the testing that are least affected by the increased need of cross-organizational communication. These areas should then be analyzed and a cost-benefit analysis should be done in which soft values such as decreased possibilities to communicate and the spreading of resources should be taken into account.

\(^{29}\)J. Bosch, *Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization*, 2002

---
5.10.3 Analysis

Having co-operation between company sites as well as other non-SEMC organizations is a good way of gaining knowledge and experience. By choosing which parts of the software engineering process is to be located outside the main site in a good way the benefits of product lines can be maintained as well as the benefits of co-operation.

5.11 Operator variability

The operator variability is considered to be out of scope in this Master Thesis since it does not reflect the product line variability. The operator variability consists of variability between SIM-cards, networks, operators and language customizations. They constitute a variability that needs to be monitored and tested thoroughly, but the extraction of the operator variability is based on operator demands and does not reflect the product line variability established by SEMC. The operator variability reflects the operator specific content and features. This variability should be monitored for each operator and tested for each product that it applies to. However, the monitoring of the operator variability should be done on a customer level, i.e. it is not directly connected to the concept of product lines and the variability between different products. This area will be passed on as future work since it is out of scope for this Master Thesis.

5.12 Sixth sense decisions

Sixth sense decision is a term that describes decisions that are made based on feeling, not backed up by statistical evidence. An example is how decisions are made regarding whether a product should be launched or not. These decisions need to be backed up with statistics showing how testing has progressed and using estimations of how many faults are still left to be found. There are simple estimation methods that can be used to enhance important decisions, some which have been found during studies at SEMC.

The sixth sense decisions are considered to be out of scope in this Master Thesis since they do not reflect the product line variability. However, the question is important and affects the overall development process greatly. This area will be passed on as future work since it is out of scope for this Master Thesis.
CHAPTER 6: CONCLUSIONS

CONTENTS:
- System test Strategy
- Generalization
6 Conclusions

The conclusions presented in this chapter have been formed based on a GQM study of how SEMC performs testing and how well the benefits of product lines are used. The study was focused on how the System Test organization at SEMC tests mother and daughter products in a specific product family. This chapter presents general conclusions about the System Test organization’s role in the engineering process at SEMC. It also presents a system test strategy with focus on taking advantage of the possibilities of re-use that product lines provide. The results from the GQM study shows that product lines save many resources and that it is a good way to maintain a growing product portfolio. In the growing market of mobile handsets it is necessary to deliver not only single products but a large product portfolio that can meet the demands of the different customer segments.

System testing at SEMC is influenced by two important factors:
1. SEMC produce embedded systems which lead to a high degree of interaction between hardware and software.
2. The software architecture at SEMC is a product line architecture using mother and daughter products to enable re-use between products.

The results from this Master Thesis confirm the need for a system test strategy that takes these two factors into account. Software and hardware interaction affects how test objects are defined at System test. This interaction is tested for the first time at system test. Product lines influences how products are defined and actualized. Both these factors put further emphasis on the need for the System test organization to have a clear product focus.

Having a product line architecture forces the development organisation to have a strict product family focus; the code is developed and tested as a part of the application software platform. In a product line environment like SEMC’s it is not until products reach system test that they exist. Figure 36 gives an overview of the four steps in a product’s development life-cycle from specification to actualization.

![Figure 36. The development life-cycle of a product.](image-url)
The concept of product lines has proved to be a good way to re-use software between projects over time. The fundamental idea of software re-use is to use older systems as the basis for new systems. This idea is applicable on different levels in a product line environment like the one used at SEMC. Figure 37 shows the different levels in SEMC’s product line architecture; the arrows indicate where re-use is possible. Creating new mother products, new daughters or new software builds can all be seen as ways of re-using old software.

![Figure 37. Different levels of software re-use in a product line.](image)

In this Master Thesis a regression test method was adapted to suit the needs of product line testing. A study was made of whether this could be used at SEMC; this is described in Chapter 5.7.1. The results imply that method designed for regression testing can be adapted to suit the needs of product line testing, enabling software re-use on a larger scale.

The fundamental idea behind testing software architectures based on re-use is to focus on changes. A new system only needs to be tested in the areas where changes have been introduced. By focusing on changes re-use can be successful during testing as well as during development. That is why the System test strategy proposed in this Master Thesis is based on a change-based test selection technique.

### 6.1 System Test Strategy

There are three basic components that influence system testing; the test object, the test specification, and selecting test cases. The software architecture used influences how the components interact. The System test strategy proposed to SEMC in this Master Thesis is based on these three components. It is configured to suit a software architecture based on

---

30 When discussing software re-use this implies not only re-use of the code itself, but re-use of all artefacts related to the software engineering process. Examples could be test re-use or re-use of requirements specifications.

product lines. It is created for an environment of large embedded systems with many products in each product family.

Figure 38. The three components that influence system testing.

Figure 38 shows the three components that influence system testing. Test selection is the most important component; it represents the action that binds the other two components together. The test selection defines how to choose which test cases to execute, based on the test object. In this thesis focus has been on change-based test selection. In change-based test selection the test object is defined as the change that has been introduced since the last build. The test specification is defined as all possible test cases defined for the entire product. The test object and the test specifications need to be well defined in order for the change-based test selection to be efficient. Together these three components are the basis of the system test strategy, but each component by itself provides a way to optimize system testing. A proposed deployment of a change-base system test strategy at SEMC is described in the following chapter.

The system test strategy proposed in this Master Thesis is also adaptable to the different product line levels. By the use of this strategy re-use can be successful between software builds, products or product families.

6.1.1 Change-based System test Strategy

The change-based system test strategy proposed in this chapter has two major advantages; the ability to establish changes of the test object between builds, products, product families and the possibility to re-use test specifications between products and product families. The strategy is described in Figure 39 where the re-use of software is marked as thick arrows between products while the re-use of test specifications is marked with thin arrows. The test selection action defines the test suite based on the test object and the test specification.
The ability to establish a test object is crucial when working with system testing in a product line environment. The test object is defined by the changes that are introduced between different builds and products in the product family. Being able to establish a test object also allows the test organisation to avoid duplicate testing of code that exists in all products in a product family.

The ability to re-use test specifications is equally important because it saves resources when developing large product families. The mapping of mother test cases to the software in the daughter products allows the test organisation to immediately identify which new test cases need to be created. The result is test specifications that efficiently test the changes that have been introduced in new products. It also diminishes the time spent on test specification development.

**Change-based test object**

The test object for the mother product consists of all parts of the product. However, the test object for daughter products consists of the changes that have been made to the mother to create a daughter. These changes can be established on several levels, defined as product variability levels. The levels and how they can be managed is further described in Chapter 5. Software variability is the most important variability to monitor, since it is in software that the greatest re-use is possible.

Having a clearly defined test object addresses the underlying issue of the need to manage product variability in a product family. While developers focus on the complete product family, system testers need to have a clear product focus; product variability needs to be
clearly defined and managed for individual products. As product families grow the
development focus must shift from individual products to the whole product families. By
having a defined and structured way of managing product variability the test object will be
easier to identify, extract and minimize. In the future precise variability management will be a
key success factor.

Re-use of the test specifications
Test specifications ensure that tested products fulfil the demands of the customer. The
traditional way to ensure this is to create test specifications based on the requirements
specification; making it possible for testers to verify that all requirements have been met. The
change-based test strategy developed in this Master Thesis proposes that the test specification
for the mother should be based on the requirements specification. The test specifications for
the daughter products should be based on the mother test specification and a coverage
analysis can ensure that this is done in a safe way.

Using a coverage analysis the test cases in the mother’s specification that exercise functions
in the daughter test object can be identified. The mother test specification can this way be
used as the basis on which the test specifications for the daughter are created. Additional test
cases can then be created to cover changes that are not exercised by the mother test
specification. This is called specification re-use, and significantly reduces the time spent on
creating daughter test specifications; it is further described in Chapter 5.8.

Coverage analysis can also provide a way to group and prioritize test cases based on how
frequently they are exercised. This grouping could lead to an organization of test cases into
real-time test cases, designed to test real-time functions, and input-based test cases. Input-
based test cases would be focused on testing functions that are triggered by user input. This is
further described in 5.8.2 and could lead to a re-organization of how system testing is
performed.

Change-based test selection
The test object defines what should be tested and the test specification defines the test cases
for the product. The test selection defines what test cases should be included in the test suite,
based on the test object. The test suite is run to ensure that no errors have been introduced in
the product. This can be done on a build, product and product family level and is explained in
more detail in Chapter 5.7.

6.2 Evaluation of strategy
After having developed the strategy described in the previous section it had to be
implemented and evaluated in the environment at SEMC. The implementation of the strategy
was done in two steps:
1. Managing variability and thereby defining the test object.
2. Mapping the test object to correct test cases.

The first step was completed by identifying six levels of variability that together form the test
object. A way of defining the software variability was achieved by creating a tool that
extracted software differences between product builds, see Chapter 5.1.2. The tool was
successful when used to establish the software variability on three levels: product builds,
products, and product families. Identifying the software differences between consecutive
product builds and directly mapping those differences to functionality was a way to improve
regression test selection. Using the tool to identify the software differences between the mother products and its daughters was also done and improved test selection when testing daughters. In general, identifying the software variability made it possible to select test cases based on changes and that was proven successful in SEMC’s organization. It enabled a more precise and safe way of selecting which test cases should be included in the test suite. This is described in more detail in Chapter 5.7.1.

The second step of the implementation of the system test strategy was to provide a safe and precise way of deciding which test cases exercise all software in the test object. In order to achieve such a mapping on a detailed level a coverage analysis was needed. A coverage analysis could also enhance re-use of test specifications between projects, as explained in the previous Chapter 5. The benefits of implementing this detailed change-based test selection strategy could be large; the technique could be used on different levels in the product line architecture and thereby save resources in many situations. However, there is a need to evaluate if the method is safe and precise in order to secure its benefits. The results from implementing the first step in the test strategy imply that a change-based test selection technique can be successful in SEMC’s system test organization. Test specification re-use could be a cheap way to improve test specification creation.

According to Bosch managing variability in a large product line environment is difficult due to the amount of products and the complexity of their configuration. This thesis has identified important variability areas that need to be monitored in order to enable efficient product line testing. A way of managing the software variability at SEMC has been suggested, evaluated and put into practice. Lastly, a way to take advantage of the detailed information on software differences has been suggested and analyzed. The results imply that the next step to take in order to improve system testing at SEMC is to implement a method to perform coverage analyses.

### 6.2.1 Limitations

There are three main limitations of the change-based system test strategy proposed in this master thesis:

- It does not take into consideration that if the product configuration changes the mapping of test cases must be re-done.
- Dynamic changes are not captured when the comparisons of software are made.
- If almost all the code is exercised by each test case the mapping of test cases to code that they exercise will become very complicated to use.

These limitations must be considered before implementing this system test strategy.

The first limitation is that if the product configuration changes a lot after the product has reached system test the benefits of this test strategy diminish. The mapping of test cases to the code that they exercise will then have to be re-done so often that a complete regression test would be more efficient.

The second limitation is that the dynamic changes that are introduced to the product will never be discovered when the product comparisons are made. If the product is dependent on dynamic variables that introduce a lot of changes these changes will not be discovered using this test strategy.
The third limitation is that if the test cases for the product are written so that each of them exercises almost all the code in the product, the mapping of test cases to code will become very complicated to use. These three factors must be considered and investigated further before this system test strategy is implemented.

### 6.2.2 Benefits

Having a clear picture of the software variability can be used to enhance several areas. Chapter 5.9 shows how product family planning can be enhanced and in 5.1 it is discussed how hidden development costs can be highlighted when analyzing software variability. In general, defining variability is a way to define the architecture of the product family which is essential to efficient product line development.

Using a coverage analysis to collect information about which parts of the code is exercised by which test cases can be used to improve other parts of testing, not only test selection. It can be used to find code that is never used i.e. code that can be removed. Using coverage information to collect statistics for individual products could be used to put focus on the test cases that exercise the code that is most frequently used, see Chapter 5.8. Different products’ end-users are decided by the marketing focus and this information together with usage information could be used to generate individual test case prioritization for each product. That way, the focus of the tests would be on testing functionality that is important to the user. Chapter 5.5 describes how marketing focus can be used to prioritize test cases.

The method proposed to perform the coverage analysis is by profiling the code. The choice of this technique is beneficial because it enables a run-time collection of data about code coverage i.e. test coverage and user coverage and automation coverage. This method is further described in Chapter 5.7.

### 6.3 Generalization

#### 6.3.1 Internal generalization

At SEMC, the change-based test strategy developed in the Master Thesis is applicable to the function test organisation. They do not have as strict a product focus as the system test organisation has but they can also benefit from monitoring where changes are introduced to focus their testing. The ability to define a test object can be useful in many different situations. The product planning organisation can use the information about the variability between products to monitor how well the product line concept has been deployed in the different product families.

#### 6.3.2 External generalization

The change-based strategy developed in the Master Thesis is applicable for companies developing large embedded systems that have a development process based on product lines. The ability to define the changes that are made to the software and focus the testing on these parts is crucial in organisations working with large product families. The strategy is most useful in an environment where the product complexity is great and software re-use is necessary to keep the development costs down.
CHAPTER 7:

FUTURE WORK

CONTENTS:

- Product family planning
- More product lines
- Improve encapsulation
7 Future Work

7.1 Product family planning

Choosing mother product
In future product families with a large amount of daughters, the mother project will have a high impact on profitability. Since the mother projects are executed before the daughter projects changes in the mother project’s time plan will affect all the daughter time plans. The task of selecting the right mother project will be influenced by many factors and the decision is vital for company success. An appropriate task for a Master Thesis would be to analyze the problem and make suggestions on how to best select which product should be chosen as the mother in a product family.

Choosing daughters
The success of product family planning is dependant on daughter products as well as mother products; their technical components, geographical location and how they relate to the mother. Financial aspects also influence family planning greatly, since release dates often are decisive of whether a product is profitable or not. Many factors need to be taken into account when choosing daughters and since it is important to ensure the success of product lines it should be analyzed thoroughly.

7.2 More product lines
The concept of product lines has proved to work well in an industrial environment. The diversity of products can be improved without increasing costs exponentially. However, the product line concept so far has been adapted mainly for saving development resources. This Master Thesis tries to investigate how product lines can save resources during system testing. However, there are still parts of the development process that may be restructured into a product line architecture similar to the software architecture. For example, the operator variability discussed in Chapter 5.11 could be restructured into a product line architecture with the possibility of switching on or off certain functionality or graphics. The different file-systems that are a part of the products could also be handled in a product line, defining common functionality for all file-systems and creating specific file-systems by adding product specific changes to that common functionality. One idea could also be to create a test specification product line where that test specifications are produced the same way as the products they should verify.

7.3 Improve encapsulation
During this Master Thesis project the need to have a well-encapsulated software has become obvious. If the system consist of individual components that can be developed and tested independently it is easier to take advantage of the product line concept. Products can be defined more clearly and development and testing can be outsourced more easily if the single components of the product line are more independent. However, a change of encapsulation changes demands on developers and on the whole software architecture. An interesting and important study would be to see how encapsulation could be improved and which effects it would have.
Reference list

Literature
Jan Bosch, Design & Use of Software Architectures, Pearson Education, Edinburgh, 2000


Articles
B. Keepence and M. Mannion. Using Patterns to Model Variability in Product Families, 1999


J. Bosch, Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization, 2002

J. van Gurp, J.Bosch, M. Svahnberg, Managing Variability in Software Product Lines

Mats Skoglund and Per Runesson, A case study of the class firewall regression test selection technique on a large scale distributed software system, 2005

M. Jaring, J. Bosch, Representing Variability in Software Product Lines: A Case Study, 2002

T. Olsson, P. Runesson, V-GQM: A feed-back approach to Validation of a GQM study, 2001
Theses
Björn Lindström, *A Software Measurement Case Study Using GQM*, 2004

Björn Pileryd & Anders Hellström, *Controlling the variant explosion*, 2005

Rickard Möller, *Optimizing Demand Paging using Profile Guided Function Positioning*, 2005

Electronic references

Nate Orenstam, http://www.valleyofthegeeks.com/Features/Cooper.html, 2006-01-10

General sources
http://www.ieee.org/portal/site

http://www.wikipedia.com
APPENDICES
## Appendices

### A Questions and metrics

Table 4. Questions and metrics

<table>
<thead>
<tr>
<th>Question</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1. What are the software differences between products in a product family?</td>
<td>Software variability.</td>
</tr>
<tr>
<td>Q2. What are the hardware differences between products in a product family?</td>
<td>Hardware variability.</td>
</tr>
<tr>
<td>Q3. What are the file system differences between products in a product family?</td>
<td>File system variability.</td>
</tr>
<tr>
<td>Q4. What are the signalling platform differences between products in a product family?</td>
<td>Signalling platform variability.</td>
</tr>
<tr>
<td>Q5. What are the marketing differences between products in a product family?</td>
<td>Marketing variability.</td>
</tr>
<tr>
<td>Q6. How can the configuration variations between products be specified on an abstract level?</td>
<td>Analysis of how product configuration is maintained by CM and the development groups.</td>
</tr>
<tr>
<td>Q7. Which test cases should be run on the mother product and on daughter products?</td>
<td>Analysis of test case selection technique used at SEMC.</td>
</tr>
<tr>
<td>Q8. How can test specifications be created so that they mirror the difference that exists in a product family?</td>
<td>Analysis of test specification development technique at SEMC.</td>
</tr>
<tr>
<td>Q9. How can the planning of what products to include in a product family be improved?</td>
<td>Analysis of Cluster planning considerations for the specified product family.</td>
</tr>
<tr>
<td>Q10. How are decisions taken about which daughter projects to locate at different site than the mother project?</td>
<td>Analysis of how many percent of the daughter projects are developed out of site.</td>
</tr>
<tr>
<td>Q11. What are the operator differences between products in a product family?</td>
<td>Operator variability.</td>
</tr>
<tr>
<td>Q12. How can difficult decisions be supported with information of product status?</td>
<td>Analysis of on what product, in the specified product family, most bugs where found and how these statistics can be used to improve future projects.</td>
</tr>
</tbody>
</table>
## B Word Index

<table>
<thead>
<tr>
<th>Word / concept</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>A failure occurs because the system is erroneous: an error is that part of the system state which is liable to lead to failure.</td>
</tr>
<tr>
<td>Failure</td>
<td>A system failure occurs when the delivered service deviates from the specified service, where the service specification is an agreed description of the expected service.</td>
</tr>
<tr>
<td>Fault</td>
<td>A fault is the cause of an error.</td>
</tr>
<tr>
<td>Product variability</td>
<td>A term describing the differences between products. Product variability is defined in three levels: product build variability, product variability and product family variability.</td>
</tr>
<tr>
<td>Product build variability</td>
<td>A term describing the differences between product builds.</td>
</tr>
<tr>
<td>Product family variability</td>
<td>A term describing the differences between product families, usually the differences between the mother products are used to describe the product family variability.</td>
</tr>
<tr>
<td>Delta</td>
<td>A factor that is used to quantify the software difference between two software builds. It is defined as: Delta = percentage changed functions + percentage added functions + percentage removed functions.</td>
</tr>
<tr>
<td>Test object</td>
<td>The object to be tested. Can be for example a complete product, a component or a list of functions.</td>
</tr>
<tr>
<td>Product family</td>
<td>In this thesis the product family describes the set of products that together form a product line. The products share functionality and are developed with a product line architecture.</td>
</tr>
<tr>
<td>Product line</td>
<td>Describes the concept of product lines i.e. the idea of developing common functionality separately and configure specific products based on that common functionality.</td>
</tr>
<tr>
<td>Product portfolio</td>
<td>The set of all products sold or developed by a company at a given time.</td>
</tr>
<tr>
<td>Alpha</td>
<td>The time when a product can be system tested. After Alpha no new functionality should be added.</td>
</tr>
<tr>
<td>Launch date</td>
<td>The date when the product is released into the market.</td>
</tr>
</tbody>
</table>