Computer Science

Stefan Lindberg and Fredrik Strandberg

The Development and Evaluation of a Unit Testing Methodology

Master's thesis

2006:3

The Development and Evaluation of a Unit Testing Methodology

Stefan Lindberg and Fredrik Strandberg

This report is submitted in partial fulfillment of the requirements for the Master of Science degree in Information technology. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Stefan Lindberg

Fredrik Strandberg

Approved, 2006-03-07

Opponent: Per Hurtig

Advisor: Donald Ross

Examiner: Tim Heyer

Abstract

Westinghouse Fuel Manufacturing in Västerås, Sweden, manufactures fuel rods for nuclear plants. Manufacturing-IT is a software development section at Westinghouse Fuel Manufacturing. This thesis involves the development of a unit testing methodology (UTM) for the Manufacturing-IT section, which currently does not follow a well-defined software test process.

By evaluating different unit testing best practices and UTM design issues collected from literature, articles, papers and the Internet, a UTM document was developed. The UTM document was developed according to requirements from Manufacturing-IT and as an extension to existing documents within the Westinghouse organization.

The UTM was evaluated by applying the methodology in a case study. A single unit within a production control system in the rod manufacturing workshop at the Westinghouse fuel factory in Västerås was tested. Asides from evaluating the UTM, the case study was intended to find software tools that could simplify the unit testing process, and to test the production control system unit thoroughly.

The 182 test cases designed and implemented revealed 28 faults in the tested unit. NUnit was chosen to be the tool for automated unit testing in the UTM. The results from the case study indicate that the methods and other unit testing process related activities included in the UTM document developed are applicable to unit testing. However, adjustments and further evaluation will be needed in order to enhance the UTM.

The UTM developed in this thesis is a first step towards a structured testing process for the Manufacturing-IT section and the UTM document will be used at the Manufacturing-IT section.

By using the methods and other unit testing process related activities in the UTM developed in this thesis, any company or individual with similar requirements for a UTM as Manufacturing-IT, and that performs unit testing in an unstructured way, may benefit in that a more structured unit testing process is achieved.

Contents

1 Introduction 1

1.1 What is a unit testing methodology? 1

1.2 Thesis summary 1

1.3 Reading guidelines 2

2 Background 5

2.1 Introduction 5

2.2 Software engineering 6

2.2.1 What is software engineering? 6

2.2.2 Principles of software engineering 7

2.3 Verification and validation 8

2.4 Software Testing 9

2.5 Unit testing 11

2.6 Unit testing at ManIT 11

2.6.1 Existing standards and documents 12

2.7 Summary 15

3 Unit testing methodology development 17

3.1 Introduction 17

3.2 Methodology design issues 20

3.3 ManIT requirements 23

3.4 Unit testing best practices 25

3.4.1 Definition of a unit 25

3.4.2 When to design and execute unit test cases 26

3.4.3 Unit testing roles 27

3.4.4 Traditional unit testing methods 29

3.4.5 Formal specification and verification 29

3.4.6 Black-box unit testing 30

3.4.7 Equivalence partitioning and boundary value analysis 30

3.4.8 The all-pairs method 32

3.4.9 White-box unit testing 34

3.4.10 Path testing 34

3.4.11 Data flow testing 36

3.4.12 Multiple condition testing 37

3.4.13 Loop testing 37

3.4.14 Testing exceptions 38

3.4.15 Unit test methods for object-oriented code 39

3.4.16 State-based unit testing ... 39
3.4.17 Selecting test cases for state-based unit testing ... 40
3.4.18 Inheritance ... 42
3.4.19 Polymorphism ... 46
3.4.20 Information hiding ... 46
3.4.21 Order of testing within a unit ... 48
3.4.22 Documentation of unit test cases ... 48
3.4.23 Unit test drivers ... 49
3.4.24 Stubs ... 50
3.4.25 Prioritizing unit tests ... 52
3.4.26 Criteria for ending unit testing ... 52
3.4.27 Redundant test cases ... 57
3.4.28 Reporting unit test results ... 57
3.5 Summary ... 61
4 Case study ... 63
4.1 Introduction ... 63
4.2 Case study expectations ... 64
4.3 Case study setup ... 64
4.3.1 Tool research ... 64
4.3.2 Test unit ... 66
4.4 Case study execution ... 68
5 Results and evaluations ... 73
5.1 Introduction ... 73
5.2 Results against case study expectations ... 73
5.3 Time estimates ... 75
5.4 Methodology usage results and evaluation ... 76
5.4.1 Traditional unit testing methods ... 76
5.4.2 Unit testing methods for object-oriented code ... 80
5.4.3 Documentation of unit test cases ... 81
5.4.4 Unit test drivers ... 83
5.4.5 Stubs ... 85
5.4.6 Prioritizing unit tests ... 85
5.4.7 Criteria for ending unit testing ... 86
5.4.8 Redundant test cases ... 87
5.4.9 Reporting unit test results ... 88
5.5 DotUnit versus NUnit ... 93
5.6 Summary ... 97
6 Conclusions ... 99
6.1 Problems ... 99
6.1.1 Learning a new programming language ... 99
6.1.2 Information hiding ... 100
6.1.3 Stubs ... 100
6.2 Conclusions ... 102
6.2.1 Conclusions based on the case study results ... 102
6.2.2 ManIT requirements ... 102
6.2.3 ManIT structured testing process ... 104
6.3 Future work ... 105

References **107**

A The concept of quality **111**

A.1 Software quality 111

A.2 Quality assurance 112

B Non-software safety measures at Westinghouse Fuel Manufacturing **113**

C Software Engineering Methodology Manual (SEMM) **115**

D Cyclomatic complexity **119**

E Flow graph notation **121**

F Code for operation LpSänd in class Rodscanner **123**

G Unit testing documentation **125**

H Test case implementation **135**

I Guidelines **139**

I.1 Guidelines for data flow testing 139

I.2 Guidelines for loop testing 140

J Definitions and abbreviations **143**

J.1 Definitions 143

J.2 Abbreviations 145

List of Figures

Figure 2.1: Unit testing at ManIT as a part of Software Engineering 5
Figure 2.2: Techniques for verification and validation 9
Figure 2.3: Westinghouse policy and procedure structure 12
Figure 2.4: Westinghouse documents relationships 15
Figure 3.1: A Classification of unit testing methods 17
Figure 3.2: The areas contributing to the development of the unit testing methodology .. 18
Figure 3.3: Software development activities described by SEMM [62] 26
Figure 3.4: General model of unit testing roles 27
Figure 3.5: ManIT model of unit testing roles 29
Figure 3.6: Traditional unit testing methods 29
Figure 3.7: Equivalence partitioning example 31
Figure 3.8: Equivalence partitioning and boundary value analysis relationship. 32
Figure 3.9: Independent paths-problematic flow graph 35
Figure 3.10: Test methods specific for object-oriented code 39
Figure 3.11: State transition diagram for a stack 40
Figure 3.12: Class B inherits class A 42
Figure 3.13: Parallel test class hierarchy 44
Figure 3.14: Pseudo code for inheriting superclass test cases 50
Figure 3.15: Continuum for how much unit testing may be done [33] 53
Figure 3.16: Relative strengths of test coverage strategies [9] 54
Figure 4.1: Overview of the case study 63
Figure 4.2: Visustin flow chart example 66
Figure 4.3: Operation hierarchy overview of class Rodscanner. 67
Figure 4.4: Case study execution steps 69
Figure 4.5: Visual Studio solution organization 70
Figure 5.1: Test case implementation for test case ID 033 84
Figure 5.2: Example of Clover.NET coverage measure results 87

Figure 5.3: DotUnit Graphical User Interface 94
Figure 5.4: DotUnit example in Visual Basic .NET code 95
Figure 5.5: NUnit Graphical User Interface 96
Figure 5.6: NUnit example in Visual Basic .NET code 96
Figure 6.1: The Visual Studio .NET solution used in the case study. 101

List of tables

Table 3.1: Relations of Chapter 3 sections. 19
Table 3.2: All-pairs matrix..... 33
Table 3.3: Response matrix for Figure 3.11 41
Table 4.1: Information exchange between the rodscanner machine and the production control system. 68
Table 5.1: Faults found during case study 74
Table 5.2: Example of an all-pair matrix used in the case study. 77
Table 5.3: Excerpt of test case documentation 82
Table 5.4: Test case design for test case ID 033. 84
Table 5.5: Faults in each impact class 90
Table 5.6: Faults found by each unit testing method 91
Table 5.7: Number of faults in each fault impact and fault type class found by each unit testing method..... 92
Table 5.8: Summary of unit testing tools features 97

1 Introduction

1.1 What is a unit testing methodology?

Firstly, the terms method and methodology are defined:

- A *method* is a systematic way of performing tasks. Systematic implies an orderly logical arrangement, usually in the form of steps [18] [66].
- A *methodology* is a documented and organized system of methods [23] [65].

A unit testing methodology (UTM) is a methodology that includes methods for performing unit testing (see Section 2.5 for details on unit testing). In this thesis a UTM will be developed and evaluated. As well as unit testing methods, the document containing the UTM developed in Chapter 3 of this thesis will also include other unit testing process related activities that are involved with unit testing.

1.2 Thesis summary

Westinghouse Electric Company is a world-wide organization that provides fuel, services, technology, plant design, and equipment for the commercial nuclear electric power industry. Westinghouse Fuel Manufacturing in Västerås, Sweden, is a part of the Westinghouse Electric Company and involves manufacturing of fuel rods for nuclear plants. Westinghouse Fuel Manufacturing includes a software development section called Manufacturing-IT. At present this section would like to improve the testing processes currently in use. A first step towards an overall improved testing process at Manufacturing-IT is the development of a UTM document. The goal of this thesis is to develop the UTM document that will describe how a unit testing process should be performed at Manufacturing-IT.

Software engineering is an engineering profession concerning all aspects of the development of software. The area of software engineering that concerns the testing of software is called software testing, and unit testing is a type of software testing. Research in general software engineering, mainly towards the area of software testing and unit testing was performed while developing the UTM. By evaluating different unit testing best practices and

UTM design issues collected from literature, articles, papers and the Internet, a UTM document was developed that included a set of unit testing methods and also other unit testing process related activities such as criteria for when to end unit testing. The UTM document was developed according to requirements from Manufacturing-IT and as an extension to existing documents, which involves unit testing within the Westinghouse organization.

The UTM was evaluated by applying the methodology in a case study. The UTM was applied on a single unit, a class named Rodscanner, within a production control system in the rod manufacturing workshop at the Westinghouse fuel factory in Västerås. The main goals of the case study were to evaluate the methods and other unit testing process related activities included in the UTM document, and to find software tools that could simplify the unit testing process. If useful software tools were found during the case study, the tools were to be integrated with the UTM. A secondary goal of the case study was to cover as many paths through the code of the class Rodscanner as possible in order to find faults.

All methods and other unit testing process related activities included in the UTM document were evaluated during the case study, except three methods for unit testing object-oriented code; inheritance, polymorphism and state-based testing, because the three methods were not applicable on the unit that was tested during the case study. The evaluations and results from the case study indicate that the methods and other unit testing process related activities included in the UTM document developed are applicable to unit testing. Two software tools that automate tests, dotUnit and NUnit, were used in the case study. By comparing and evaluating the two software tools, NUnit was selected as the superior. Visustin, Project Analyzer and Clover were other useful software tools used during the case study.

Higher coverage of the code in the class Rodscanner was reached than was expected by the case study performers. 182 test cases were designed and implemented using the UTM. The test cases revealed 28 faults. Conclusions based on the thesis work, case study and results are included in Chapter 6.

1.3 Reading guidelines

To fully understand the contents of this thesis, the reader is recommended to at least have basic knowledge in the area of computer science. Before reading through this thesis, the reader is recommended to read through the definition list and the abbreviation list, provided in Appendix J.

Important to note is that:

- When the term ‘testing strategy’ occurs in quotes, ‘strategy’ may be interpreted as ‘methodology’.
- For class methods in object-oriented programming, the synonym ‘operation’ is used, so that there is no confusion with the term ‘method’ used in the context of the methodology. When the term ‘method’, referring to the operation of a class, occurs in quotes, the construct [class] is inserted before ‘method’, i.e. “… [class] method …“.

The reader is encouraged to read the chapters in an ascending order. The thesis is structured as follows:

- *Chapter 2: Background*.

  Chapter 2 describes unit testing in a software engineering perspective, unit testing at Manufacturing-IT and defines the goal and purpose of the thesis work.
- *Chapter 3: Unit testing methodology development*.

  Chapter 3 summarizes and evaluates the material used in the UTM research. Concludes which methods and other unit testing process related activities should be included in the UTM document.
- *Chapter 4: Case study*.

  Chapter 4 describes how the case study was set up and executed. Expectations on the results from the case study are also included in the chapter.
- *Chapter 5: Results and evaluations.*

  Chapter 5 presents the results from the thesis work, including the case study, and evaluates the results.
- *Chapter 6: Conclusions.*

  Chapter 6 describes the problems encountered during the thesis work, final conclusions and future work.
- *Appendix A*: *The concept of quality.*

  Appendix A introduces and defines the terms quality and quality assurance in the context of software.
- *Appendix B: Non-software preventive safety measures at Westinghouse Fuel Manufacturing*.

Appendix B describes preventive measures that have been taken at Westinghouse Fuel Manufacturing in order to assure safety in case of software failures.

- *Appendix C: Software Engineering Methodology Manual (SEMM).*

  Appendix C summarizes the internal Westinghouse document SEMM.

- *Appendix D: Cyclomatic complexity.*

  Appendix D introduces and explains cyclomatic complexity.

- *Appendix E: Flow graph notation.*

  Appendix E introduces the flow graph notation.

- *Appendix F: Code for operation LpSänd in class Rodscanner.*

  Appendix F includes the Visual Basic .NET code for the operation LpSänd in the class Rodscanner, which was tested in the case study.

- *Appendix G: Unit testing documentation.*

  Appendix G includes examples of documentation of test cases, flow graphs, truth tables, fault reports and unit test reports.

- *Appendix H: Test case implementation.*

  Appendix H includes examples of the implementation of test cases for the operation LpSänd in the class Rodscanner.

- *Appendix I: Guidelines.*

  Appendix I includes guidelines that describes how data flow testing and loop testing should be performed.

- *Appendix J: Definitions and abbreviations.*

  Appendix J explains and defines terms and abbreviations used in this thesis.

2 Background

2.1 Introduction

The software development section; Manufacturing-IT (referred to as ManIT in the rest of this thesis), at Westinghouse Fuel Manufacturing in Västerås, Sweden, does not follow a well-defined procedure that concerns testing of the software being developed. The goal of this thesis is to develop a UTM document that will describe how a unit testing process should be performed at ManIT. The document will lead to improved unit testing at ManIT and will be a first step towards an overall improved testing process at ManIT.

Figure 2.1 shows both the outline for this chapter and unit testing at ManIT as a part of software engineering. The figure indicates that unit testing is a part of software testing, which in turn is a part of verification and validation, which is a part of software engineering. By applying unit testing theory and unit testing best practices, the unit testing at ManIT may be improved.

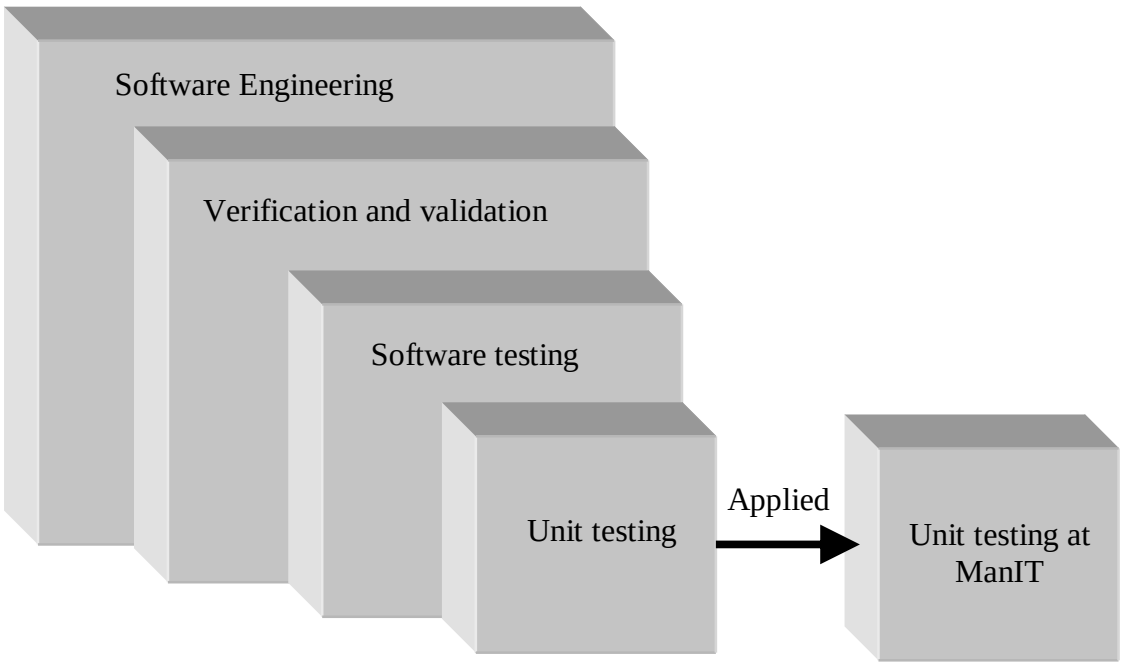


Figure 2.1: Unit testing at ManIT as a part of Software Engineering

This chapter will cover each of the areas named in Figure 2.1, namely:

- Software engineering
- Verification and validation
- Software testing
- Unit testing
- Unit testing at ManIT

Software Engineering in general and its different principles will be discussed in section 2.2. Section 2.3 will cover verification and validation, followed by sections 2.4 and 2.5, which describe software testing and unit testing from a general perspective respectively. Section 2.6 will cover unit testing at ManIT from a more critical perspective. “How is the unit testing performed at ManIT today?”, “What problems with the unit testing process exist?” and “Are there any existing standards and documents that specify how unit testing should be done?” are questions that will be answered in Section 2.6.

2.2 Software engineering

2.2.1 What is software engineering?

Sommerville [50] states that “Software engineering is an engineering discipline which is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.” This definition implies that software engineering concerns all the traditional software development phases: definition, development and maintenance.

Pressman [46] includes a definition of software engineering by the IEEE [24]:

(1) Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
(2) Software engineering is the study of approaches as in (1).

This definition, from IEEE, is similar to the definition given by Sommerville [50]. But the IEEE also introduces the aspect of studying the approaches of what software engineering is.

Software engineering concerns all aspects of the production of quality software[1]. To produce quality software, methods, tools and processes are used.

Software engineering has a number of principles. In the subsequent sections some of the principles will be covered, mainly the principles associated with software testing.

2.2.2 Principles of software engineering

Pressman has eight activities that he calls "umbrella activities" [46]. These activities may be directly translated as principles. Not all principles are concerned with software testing, therefore only the principles essential to software testing will be covered in this section.

1. *Software quality assurance*

   A definition and discussion of "software quality" and "quality assurance" is given in appendix A.

2. *Document preparation and production*

   Documentation is an important aspect of software engineering. Staff may read documentation, track what has been done and not been done, and track progress. When members of the staff document their work they also increase their knowledge of the work that has been performed. Documents should be prepared and produced in standard ways to provide structure and readability. Test documentation should be a significant part of a successful testing methodology.

3. *Measurement*

   Measurement of processes, products and staff activities may provide essential facts about how to improve different aspects of the software development process and the organization. Measurement of tests and quality factors may contribute to improve a testing process. The only measurement included in the scope of this thesis is the measurement of faults found when using the UTM. The faults found in the case study will be counted and categorized to aid in the evaluation of the UTM.

4. *Risk management*

[1] What is meant by "quality software" is discussed in Appendix A.

Any development of software includes risks. The risks have to be identified and may be categorized depending on the threat associated with each risk. Identifying risks may contribute to prioritizing tests, which may eliminate the most critical risks.

2.3 Verification and validation

Verification and validation are the processes of checking and analyzing whether a software product conforms to its specification and meets the needs of the users [50]. The difference between the two may be described by the following two questions [25] [50]:

Verification: “Are we building the product right?”
Validation: “Are we building the right product?”

Thus, verification is about conformance to the specification, and validation is about the needs of the users.

Three important terms that will be used throughout this thesis are (based on [11]):

- A *fault* is missing or incorrect specification, design or code. The term fault also covers “defect” and “bug”.
- An *error* is a human action that produces a fault.
- A *failure* is the manifested inability of a system or component to perform a required function within specific limits.

A fault may or may not result in a failure. There may for instance be a fault in a section of code that is never executed, thus not resulting in a failure.

It is also important to note that verification and validation is about establishing the existence of faults, whereas the process of locating and correcting faults is called debugging [50].

Figure 2.2 shows two different techniques for Verification and Validation [50]:

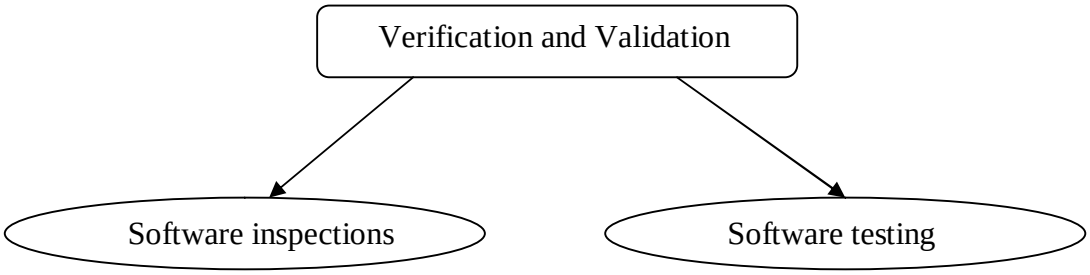


Figure 2.2: Techniques for verification and validation

Software inspections consist of inspecting and reviewing design and code documents with the intention of finding faults. Software inspections are sometimes included in software testing as static software testing [43].

Software testing concerns the execution and simulation of software programs with the purpose of finding faults in a program [16].

2.4 Software Testing

Software testing is an activity concerned with uncovering evidence of faults in software. Software testing is also concerned with ensuring that the product meets the requirements from the customers and the users. The software engineer tests to find out if anything that may go wrong does go wrong. “Testing is a process of executing a program with the intent of finding an error.” [16] In other words, “Testing is important because it substantially contributes to ensuring that a software application does everything it is supposed to do.” [33]

One of the reasons that software development projects fail is insufficient and/or inadequate testing [28]. Therefore, software testing should be a part of every software development process in order to increase the probability of successful projects.

Software testing is often equated with quality assurance but software testing is only a part of quality assurance.[1]

Software testing is traditionally an activity performed after analysis, design and implementation in software development. But the more the testing process is integrated with other activities in the process of developing software, the better [25].

[1] What is meant by “quality assurance” is discussed in Appendix A.

There are a number of different software testing types, e.g. unit testing, integration testing, system testing and user acceptance testing. In this thesis, unit testing will be covered.

Unstructured software testing means that there are no formal standards, procedures or guidelines that are being used when it comes to testing the software. Why is it better to test according to standards than to test by intuition? What are the advantages of a structured software testing approach? According to Martin Pol et al. [45], a structured testing approach offers the following advantages:

- Testing gives insight into and advice about the risks regarding the quality of the tested system.
- Faults are found at an early stage.
- Faults may be prevented.
- Testing is performed in as short a time as possible on the critical path of the total development cycle, so the development time will be shorter.
- The test process is transparent and manageable.

A structured software testing process provides a level of control over the testing process. An improved testing process contributes to a higher rate of fault detection and better evaluations of the quality of the software.

Sommerville explains some reasons why software standards are important [50]:

- The standards suggest what is the best, or at least most appropriate, practice. This knowledge is often acquired after a great deal of trial and error. Using a standard avoids repeating the same mistakes as in the past.
- Standards provide a framework for quality assurance. Quality control is then an activity to make sure the standards have been followed.
- Work carried out by one person is continued by another. No need to reinvent the wheel.
- Standards make all engineers within an organization use the same practices.

Standards may be used as a means for improvement of the testing process through comparisons, metrics and best practices. Activities such as planning, estimating and resource handling will become more efficient when using standards.

2.5 Unit testing

Unit testing is the testing of a single unit, separated from other units of the program [16]. Observe that the unit is only separated from other units during the unit testing process. Sometimes, when the units are strongly interdependent, a unit must be tested together with all the units that the first unit depends on [29]. A unit may for instance be a function/procedure or a data collection. In object-oriented development a unit is normally a class or an interdependent cluster of classes [11] [54].

Unit testing is commonly performed by the developer during the implementation activity of a software development process.

Hutcheson [21] claims that testing the lower levels of a system (i.e. unit testing) requires highly technical testers and that low-level testing is a difficult, time-consuming and expensive task. Further, Hutcheson [21] claims that less and less low-level testing is done in top-down approaches to software development, arguing that "if the system seems to give the correct response to the user, why look any further?" In favor of unit testing, Hutcheson finally recognizes that "the problem is that the really tough, expensive faults often reside in the lower-level areas" [21]. Another argument for performing unit testing is that it localizes the faults found to the unit tested. If unit testing is not performed, faults found later during integration testing may lead to time-consuming work because there are too many places to look for the faults [41].

2.6 Unit testing at ManIT

The current testing process, including unit testing, at ManIT in Västerås is unstructured. Testing has not been a significant part of the development at ManIT and testing has been performed in the sense that the programmers "test what they feel is right to test", i.e. in an informal manner. Also, the test documentation has been poor or non-existing.

In Section 2.6.1, about existing documents in the Westinghouse organization, a number of documents that describe how the testing processes at the company shall be performed are mentioned. The problem with these documents, according to ManIT, is that they are neither specific nor detailed enough. ManIT requires a well-defined and easy-to-follow document(s) that describes how testing should be performed.

2.6.1 Existing standards and documents

There are a number of existing documents in the Westinghouse organization that describe standards for testing procedures. The organization policy and procedure standards may be described with the following figure:

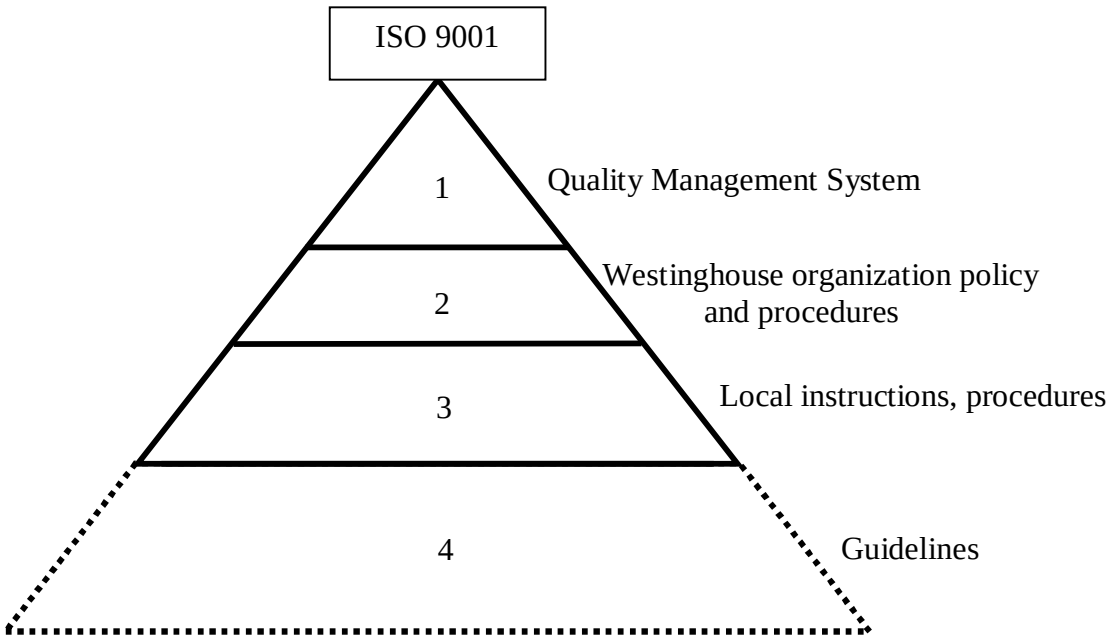


Figure 2.3: Westinghouse policy and procedure structure

Westinghouse is an ISO 9001 certified company. ISO 9001 pervades all activities in the Westinghouse organization. ISO 9001 is a comprehensive standard written at a high level of abstraction, and needs to be described in more specific ways. Therefore, Westinghouse has produced a number of documents that are based on ISO 9001. The documents explain in more specific ways how activities shall be performed to assure quality. The levels of the pyramid in Figure 2.3 are all based on the top of the pyramid, ISO 9001. The documents in the lower levels are more specific than in the higher levels. As the documents get more specific, the more documents are required. Level 4 is an “informal” level, in the sense that the level is not defined in the Westinghouse organization. Level 4 is used because more specific documents than the documents in level 3 are needed. In the following sections, descriptions of the documents in the Westinghouse organization that involve software testing are provided.

2.6.1.1 ISO 9001

Eklund and Fernlund [16] write that ISO-9001 is the complete quality standard in the ISO-9000 standard family. They further explain that the ISO 9000-3 is a guidance to the ISO-9001 concerning development, delivery and maintenance of software [16]. These documents are at the highest level of the pyramid (Figure 2.3) and influence all lower level documents and standards. They will not be discussed further, as those aspects of the ISO-9000 standard that are relevant for this thesis will be reflected in the lower level documents.

2.6.1.2 Quality Management System

The Quality Management System (QMS) [61] is a document that describes the Westinghouse commitments to the quality assurance requirements of, mainly, ISO 9001.

The Westinghouse QMS comprehensively describes different aspects of how quality assurance within Westinghouse shall be conducted. There are three major areas that the QMS incorporates [61]:

1. Quality planning.
2. A framework for managing the activities that enable the company to create items and services which consistently satisfy the customer and regulatory requirements.
3. Achievement of enhanced customer satisfaction.

The area that concerns this thesis is point number two in the list above. Therefore, this thesis does not discuss the other two areas.

The QMS generally describes quality assurance processes for all engineering disciplines within the Westinghouse organization. In other words, the QMS does not describe specifically how software engineering should be performed to assure quality in a product. But it does give general indications. As an example, the QMS document explains that all software development activities should be documented, and the requirements of QMS shall be used in all organizations that develops or supplies computer software [61].

The QMS also describes, in a general way, how analysis, testing, measurement and improvement of items (in the case of this thesis; software) and processes are intended to be performed.

The QMS is a high level document that comprehensively describes quality assurance and organization improvement. In the next sections EP-310 and the Software Engineering

Methodology Manual will be covered, which are both based on the QMS in the direction of software engineering.

2.6.1.3 EP-310

EP-310 (Engineering Procedure 310 [59]) is a procedure that is on level 3 in the Westinghouse policy and procedure structure (Figure 2.3). EP-310 will be applied at the ManIT department in Västerås from the start of 2006. The purpose of EP-310 is to describe "the design and development, software verification and validation, maintenance and control of Computer Software in Nuclear Fuel Engineering" [59]. It is based on the QMS, and applies the generally described quality management of the QMS on software engineering. There are however other procedures also describing aspects of quality management on software engineering. These documents are referred to within EP-310.

2.6.1.4 SEMM

SEMM is an abbreviation of "Software Engineering Methodology Manual" [62]. This procedure has a more "nuts and bolts" approach to software development than EP-310. SEMM is on level 4 in the Westinghouse policy and procedure structure (Figure 2.3) and is a large document on 400 plus pages based on EP-310. SEMM is the lowest level procedure available. SEMM is very specific on some areas, e.g. coding standards, but less specific on other areas, e.g. testing; thus the need for a more specific testing methodology document. Just like EP-310, SEMM will be applied at ManIT from the start of 2006.

Parts of both EP-310 and SEMM are specific to certain software or tools. These parts will not be adopted at ManIT. Examples of specific parts in EP-310 and SEMM are the use of the operating system UNIX and the version control system ClearCase in SEMM. Since ManIT uses other systems (Windows and SourceSafe), the detailed guidelines in SEMM are not applicable.

SEMM does not include much about unit testing specifically. For a brief summary of relevant parts of SEMM, see Appendix C.

The UTM developed in this thesis will be at the same level in the hierarchy (Figure 2.3) as SEMM, which makes SEMM the most relevant document to base the UTM on. The EP-310, QMS and ISO-9000 documents will not be further consulted, since SEMM contains the specific information about testing relevant for this thesis.

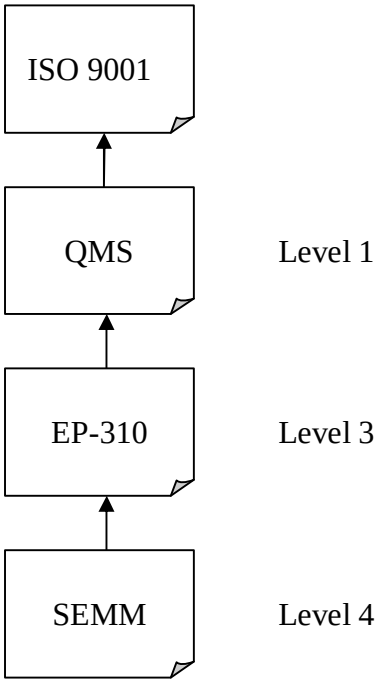


Figure 2.4: Westinghouse documents relationships

The levels showed in Figure 2.4 are the same levels as in Figure 2.3. Each document in Figure 2.4 is located at the corresponding level (There is no document concerning software testing at level 2). The arrows indicate that QMS is based on ISO 9001, EP-310 is based on QMS and SEMM is based on EP-310. The UTM developed in this thesis will act as an extension of SEMM in the area of unit testing.

2.7 Summary

Software engineering is an engineering profession concerning all aspects of the development of software. Verification and validation is a major part of software engineering and consists of checking and analyzing whether or not a software product conforms to its specification and meets the needs of the users. One activity for verifying and validating software is software testing. Software testing is an activity that involves the uncovering of evidence of faults. Unit testing is a type of software testing and is the testing of a single unit, separated from other units of the program.

The ManIT department at Westinghouse does not follow a structured software unit testing process. By developing and using a UTM, which applies software engineering principles and best practices that are specific for unit testing, the unit testing process at Westinghouse will be improved. The UTM document will extend existing standards and documents at Westinghouse that are concerned with software testing.

3 Unit testing methodology development

3.1 Introduction

In this chapter, a methodology for unit testing will be developed. One of the requirements from ManIT, which will be described in Section 3.3, is that the UTM should be applicable when unit testing object-oriented software systems. Consider Figure 3.1 that illustrates a

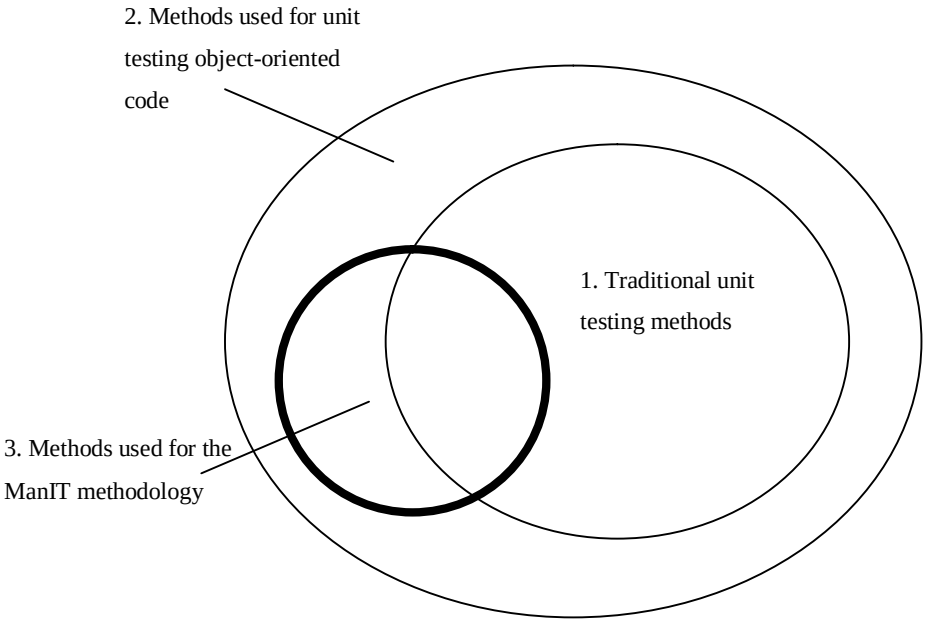


Figure 3.1: A Classification of unit testing methods

classification of unit testing methods. Area one, 'Traditional unit testing methods' represents traditional methods that may be used for both unit testing object-oriented code [13] as well as unit testing more "traditional" code, such as procedural or functional code. Area one is included in area two, 'Methods used for unit testing object-oriented code', which represents all methods that may be used when unit testing object-oriented code, illustrating that for object-oriented code, there are additional unit testing methods to consider. Area three, 'Methods used for the ManIT methodology', represents the methods that will be used for the

ManIT UTM developed in this chapter. Thus, the UTM will include traditional methods, as well as methods that are specific for unit testing object-oriented code.

In this chapter, different methods for unit testing will be discussed and the most suitable methods will be adopted in the UTM. Other unit testing process related activities that are not directly classifiable as methods, such as the roles in unit testing, will be discussed and, if suitable, included in the UTM document. The methods included and other unit testing process related activities will result in a preliminary UTM document. The preliminary UTM will be evaluated using a case study, which will provide useful feedback, resulting in possible revision of the methodology. Figure 3.2 shows the UTM together with the areas that contribute to the development.

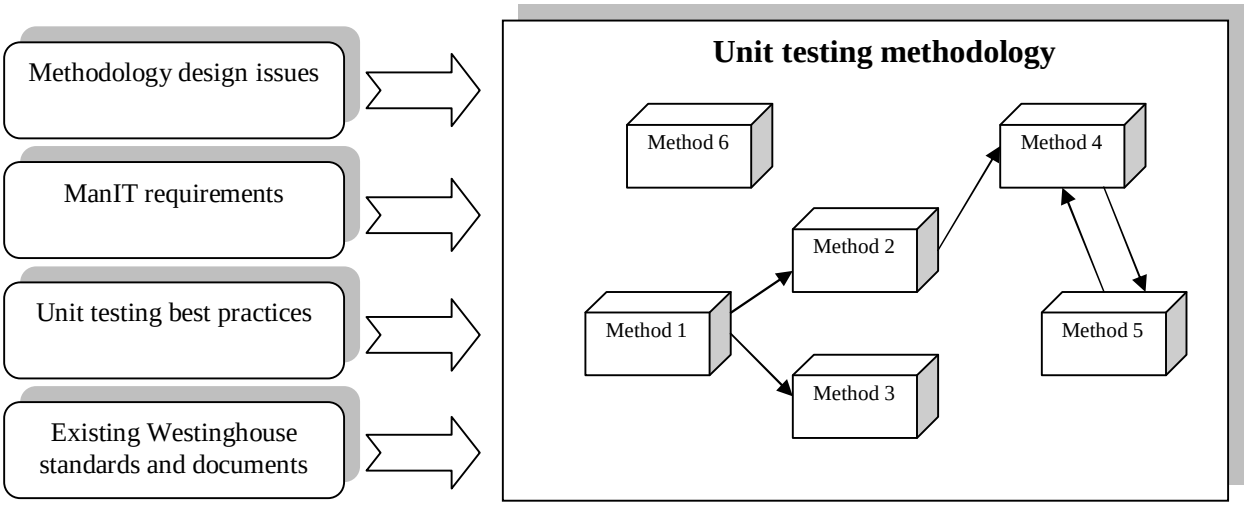


Figure 3.2: The areas contributing to the development of the unit testing methodology

The contributing areas are:

- *Methodology design issues*. Will be treated in Section 3.2.
- *ManIT requirements*. Will be treated in Section 3.3.
- *Unit testing best practices.* The best practices are ways proven through experience of doing unit testing related activities. Agreement among different references to literature, papers, articles or web pages will be considered best practices. Some unit testing methods will be considered best practices. Best practices will be treated in Section 3.4.
- *Existing Westinghouse standards and documents.* An example of an existing Westinghouse document is SEMM. This area was treated in Section 2.6.1.

The UTM consists of a number of methods. The arrows in Figure 3.2 indicate that the methods are related to each other. The relations may for instance be that Method 1 must be completed before Method 2 and Method 3 may be started, i.e. time dependence. Another type of relation could be that Method 4 and Method 5 depend on each other in such a way that one of the two should not be performed without the other.

Section 3.4, “Unit testing best practices”, will include sections for different unit testing methods and other unit testing process related activities. The sections starting from Section 3.4.4, “Traditional testing methods”, to Section 3.4.22, “Documentation of unit test cases”, are related according to Table 3.1. As these different methods and other unit testing process

Traditional testing methods
Formal testing
Black box
Equivalence partitioning and boundary value analysis
The all-pairs method
White box
Path testing
Data-flow testing
Multiple condition testing
Loop testing
Exception testing
Methods for testing o-o code
State-based unit testing
Selecting cases for state based testing
Inheritance
Polymorphism
Information hiding
Unit testing aspects
Order of testing within a unit
Documentation of unit test cases

Table 3.1: Relations of Chapter 3 sections.

related activities are considered for the UTM document, requirements and recommendations from the affecting areas in Figure 3.2 will be discussed and used to determine conclusions.

3.2 Methodology design issues

When developing a UTM a number of design issues should be considered. The methods included in the UTM will address different design issues.

McGregor and Sykes [33] explain that there are many activities that comprise testing. With respect to these activities, five dimensions of testing are identified. These dimensions may be described as answers to the following questions:

- Who performs the testing?
- Which pieces will be tested?
- When will testing be performed?
- How will testing be performed?
- How much testing is adequate?

These questions may be compared to Kaners et al. [26] five dimensions of testing:

- "*Testers.* Who does the testing."
- "*Coverage.* What gets tested."
- "*Potential problems.* Why you're testing."
- "*Activities.* How you test."
- "*Evaluation.* How to tell whether the test passed or failed."

Kaner et al. [26] explain that all testing involves these five dimensions. Therefore, methods included in a UTM should together cover all the five dimensions. Design issues for the methodology covering both the five dimensions described by McGregor and Sykes [33] and Kaner et al. [26] will be:

- Unit testing roles
  (Who does the testing?)
- When to design and execute unit test cases
  (When will testing be performed?)

- Criteria for ending unit testing
  (How much testing is adequate?)
- Traditional unit testing methods/Unit test methods for object-oriented code
  (Which pieces will be tested? How to test.)
- Order of testing the units
  (How to test.)

Kaner et al. [26] point out three basic questions that should be asked about a test methodology:

1. "Why bother?
   Testing is expensive. Don't include activities in your strategy unless they address a risk that matters enough to spend time testing."
2. "Who cares?
   Reasons to test are not laws of nature; they're rooted in the feelings and values of people who matter. Don't include activities in your strategy unless they serve somebody's interest."
3. "How much?
   Some strategies are much easier to say than do. 'We will test all combinations of printer features' is one short sentence that launches a thousand tests (or a hundred thousand). How much of that are you really going to do?"

These three questions state additional issues that need to be considered when developing a UTM. The design issue described by point number one will be covered in a section called "Prioritizing unit tests". Considering point number two in the list, interests from ManIT and other involved staff at Westinghouse will be decisive when including different methods in the UTM. Point number three has already been considered in this section ("how much testing is adequate?" [33], details will be provided in Section 3.4.26).

Kaner et al. [26] include a number of different issues that need to be considered when developing a methodology. Three major categories describe choices that must be made about the test process [26]:

- "*Strategy*. How will you cover the product to find important problems fast? What specifically will you test? What techniques will you use to create tests? How will you recognize bugs when they occur? The test strategy specifies the relationship between the test project and the test mission."
- "*Logistics*. How will you apply resources to fulfill the test strategy? Who will test? When will they? What do you need to be successful?"
- "*Work products*. How will your work be presented to your clients? How will you track bugs? What test documentation will you create? What reports will you make?"

A number of the issues included in these three categories have already been addressed in this section. Additional issues, covering the above three categories, are:

- Classification of faults
- Documentation of unit test cases
- Documentation of unit test results

Kaner et al. [26] explains what a good testing methodology should be:

1. *Product-specific.* A generic testing methodology may be good but a methodology that is specific to the product will be better.
2. *Risk-focused.* Focus the test process on the most important aspects of a project.
3. *Diversified.* A varied methodology that includes different test approaches and methods.
4. *Practical.* A performable methodology. If the methodology is beyond the capabilities of projects, the methodology should not be suggested.

The UTM developed in this thesis will be specific to the requirements of ManIT. One requirement (see Section 3.3) is that the UTM should be applicable to different categories of projects and environments at ManIT.

Point number two has already been considered in this section and details will be provided in Section 3.4.25.

Concerning point number three, it is better to have a number of different alternative test approaches that are at a reasonable good level than to have one or two approaches that are perfect [26].

Point number four indicates that the UTM should be performable. A case study that will include usage of the methodology will be included in this thesis to ensure that the methodology will be performable.

Kaner et al. [26] state that "for best results, you should be in control of your testing, not your documentation." This implies that documentation, like the UTM developed in this thesis, should not over-specify what the tester should do. The methodology should leave room for tester creativity that is an important issue to consider in the development of the methodology. The UTM should encourage the tester to generate ideas and see possibilities. It is the tester that will write the unit test cases, not the methodology.

Is it unwise to introduce several methods at once in an organization? Bell et al. [10] argue that new methods should be introduced one at the time, for two reasons:

1. If too many new approaches are adopted at once, it is hard to determine which of them are being effective.
2. If too many of the existing methods are destroyed, the morale of the organization might be threatened.

Pol et al. [45] describe that implementing change in too many large steps may result in failure and that the staff fall back to old unchanged behavior. Pol et al. [45] further write that "[…] change should be conducted gradually and in a controlled manner" [45]. All methods in the UTM should not be adopted at once, but gradually. Some methods might have a strong dependence on each other and therefore must be used together, but others may be used independently. It is therefore possible to adopt the UTM step by step.

3.3 ManIT requirements

The following bullet list contains requirements defined by the development environment and the staff at ManIT:

- Focus on and compatibility of the .NET environment

The development at ManIT is and will be performed using the .NET environment. The UTM should therefore be applicable when developing in .NET. .NET contains a framework of classes, and is an environment in which the developer may build, create and deploy software applications. Restrictions on the UTM set by the .NET environment will be found during the case study (described in Chapter 4), if there are any restrictions.

- Unit testing object-oriented software systems

  The software development at ManIT is and will be performed using object-oriented development. The UTM should therefore be applicable when unit testing object-oriented software systems.

- Generic methodology

  The UTM should be applicable in as many situations as possible at ManIT. Pol et .al [45] state that an organization should use a testing methodology that is sufficiently generic to be applicable in every situation, but that contains enough detail so that it is not necessary to reinvent the same methods for each project.

- Automated execution of unit tests

  It should be possible to automatically execute the unit tests. Automatic execution of unit tests are supported by testing tools, for instance by NUnit [39].

- Effect of the development group size

  The software development group at ManIT is currently very small. Does the size affect the design of the UTM, and if yes, how? The following arguments are intended for software development processes in general, but since unit testing is a part of the development process, the discussion applies to unit testing as well. Sommerville [50] states that for small projects, where there are only a few team members, the team may not spend a lot of time on tedious administrative procedures, and good development technology is particularly important. Jacobson et al. [25] argues that even for a small project done by a small team, detailed instructions of work routines are necessary. Otherwise “project members are left to

solve problems as they arise, with bad work routines, inferior systems and project delays as a result." [25]

Thus, the UTM should not contain too many administrative procedures, such as writing and updating various documents. Any such procedures should be aided by technology, i.e. software tools.

3.4 Unit testing best practices

Recall from Section 1.3 that:

- When the term 'testing strategy' occurs in quotes, 'strategy' may be interpreted as 'methodology'.
- For class methods, the synonym 'operation' is used, so that there is no confusion with the term 'method' used in the context of the methodology. When the term 'method', referring to the operation of a class, occurs in quotes, the construct [class] is inserted before 'method', i.e. "… [class] method …".

3.4.1 Definition of a unit

Remember the definition of unit testing in section 2.5: "Unit testing is the testing of a single unit, separated from other units of the program." [16] In procedural languages the units are functions or procedures [54]. This suggests that the operations of a class in object-oriented programming are candidates for the units to test. This is however not a good idea since "[class] methods are meaningless when separated from their class and treated in isolation. For example, a [class] method may require an object to be in a specific state before it can be executed, where that state can only be set by another [class] method (or combination of [class] methods) in a class" [54]. The best candidates are instead a class [13] [46] [54] or a small cluster of interdependent classes [11] [13].

As Binder [11] notes, the difference between integration and unit testing is not as clear in object-oriented testing as in procedural testing. When testing a cluster of classes, these classes are also integrated.

The 'best practice' is that the basic unit for unit testing in object-oriented development is the class. Sometimes a cluster of classes must be unit tested together, if the classes "are so tightly coupled that testing constituent classes in isolation is impractical" [11]. The focus in

this thesis will primarily be on unit testing individual classes, but in some situations on clusters of classes, e.g. inheritance hierarchies (see Section 3.4.18 for details).

3.4.2 When to design and execute unit test cases

There are three possible moments, in the software development process described by SEMM [62] (see Figure 3.3), when the unit test cases may be designed:

1. During the test planning activity.
2. During implementation activity.
3. During the formal testing[1] activity.

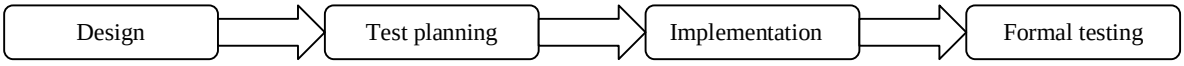


Figure 3.3: Software development activities described by SEMM [62]

SEMM [62] provides two alternatives for when unit tests may be executed:

1. During the formal testing activity.
2. During the implementation activity.

If the unit tests are to be executed during the formal testing activity the unit test cases should be designed during the test planning activity [62]. If unit tests are to be executed during the implementation activity, SEMM [62] provides no particular description about when the unit test cases should be designed.

There is agreement that the test cases should be designed before the unit code is implemented [7] [8] [13] [16] [29]. Designing test cases before implementation of the unit code "results in smaller classes and methods and better interfaces" [13]. The previous two arguments indicate that the best choice for when test cases should be designed is during the test planning activity.

[1] Not to be confused with formal specification and verification that involves mathematical proofs. SEMM states that "the purpose of the formal testing activity is to perform the testing as identified in the Test Plan" [62]. It is however important to note that for unit testing, SEMM provides the choice to perform the unit tests during the implementation activity.

There is a possibility that all test cases needed for a unit are not found during the test planning activity, e.g. because of shortage of unit design or unit specification. Especially white-box test cases may be difficult to design because knowledge of how the unit will be implemented is required. It should therefore be allowed to design additional, missing test cases whenever they are identified during the implementation activity.

If the test cases are executed during the implementation activity, the test cases may provide fast feedback, which is desirable [8].

The conclusion is that primarily, the test cases needed for a unit should be designed during the test planning activity. Additional, missing test cases may be identified and designed during the implementation activity. Tests should be executed during the implementation activity.

3.4.3 Unit testing roles

As a starting point for the discussion in this section consider the general model of unit testing roles shown in Figure 3.4 containing the following three roles:

- Unit test case designer
- Unit test case executer
- Unit implementer

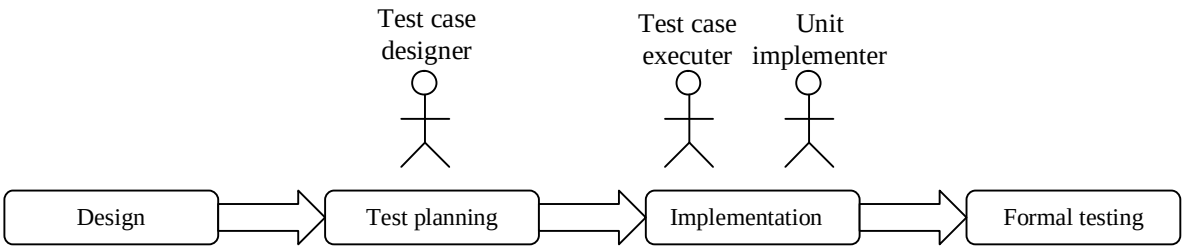


Figure 3.4: General model of unit testing roles

The person that implements a unit is also the person testing the unit [21] [28]. According to Jacobson et al. [25], the main reason for letting the unit implementer test the unit is due to cost. Lindegren [29] states that it is a common belief that the same person should never implement and test the same unit. This belief however, only applies if the test cases are written after the code, since the unit implementer then tends to be affected by the particular coded solution when designing test cases [29]. If instead test cases for a unit are designed before the implementation of the unit, as in the conclusion of the previous section, unit testing

becomes more effective if the unit implementers design the test cases [29]. Additional benefits are, according to Lindegren [29], that the unit implementer is forced to understand the specification of the unit and that the unit implementer will learn to design better test cases.

According to McGregor and Sykes [33] there is a disadvantage with having a person that implements a unit also design test cases for that unit. If the unit implementer misunderstands the specification of the unit, this misunderstanding will lead to faults in both the test cases and the unit implementation [33]. An inspection[1] process may be used to detect faults in test case designs [45]. The inspection should prevent the possible misunderstanding of the specification by the unit implementer. Considering the development group size at ManIT, people for an inspection may not be available. Therefore, test cases may be inspected by any available person in the ManIT staff, e.g. a developer partner. Having a developer partner inspect test cases is better than having no inspection at all, considering the motivation for inspections. Also, certain software development process specific activities may help avoid the problem with misunderstanding the specification described by McGregor and Sykes. For instance, when developing according to eXtreme Programming all development is done in pairs, and there is a focus on communication [8]. Pair programming and a focus on communication help to avoid the problem with misunderstandings of the specification.

The conclusion is that one person should have all three roles; unit test case designer, unit test case executer and unit implementer (see Figure 3.5). The conclusion is based on the advantages with the approach of one person having all three roles discussed in this section. The inspection process described by SEMM [62] (see Appendix C) will prevent specification misunderstandings to be implemented.

[1] Inspections are not a subject included in this thesis but are described thoroughly in SEMM [62] (see Appendix C for more details)

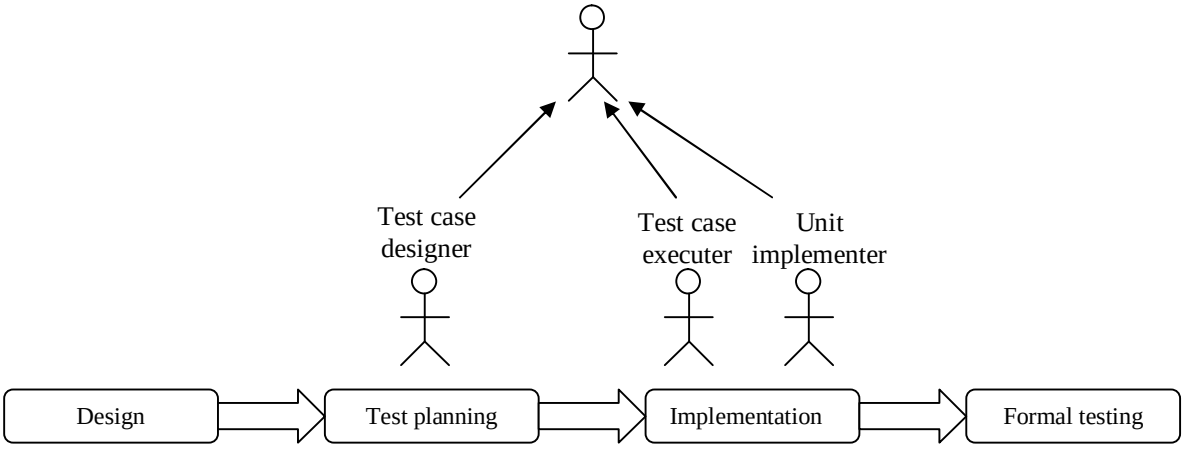


Figure 3.5: ManIT model of unit testing roles

3.4.4 Traditional unit testing methods

The methods discussed in sections BLA to BLA are traditional unit testing methods, which are represented by the shaded area in Figure 3.6 (derived from Figure 3.1 in Section 3.1). The unit testing methods are called tradtitional in the sense that they are used to test “tradtitional” code such as procedural or functional code. The methods may also be applied when testing object-oriented code.

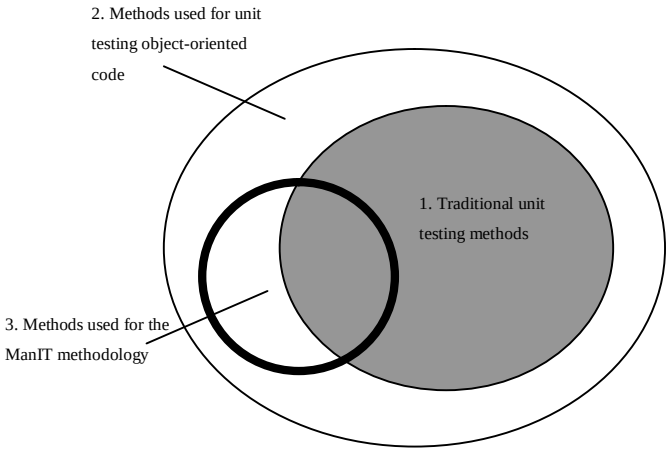


Figure 3.6: Traditional unit testing methods

3.4.5 Formal specification and verification

Formal specification and verification consist of testing methods that use a mathematical notation for the specification of the software system, and then by mathematical deduction verify the system. Formal specification and verification, using for example such notations as Z [55], is best suited for safety-critical or real-time systems [50]. ManIT has decided that,

although interesting, formal specification and verification are not in the scope of this thesis (see Appendix B for some of the non-software safety measures at Westinghouse Fuel Factory that justifies not using formal specification and verification).

3.4.6 Black-box unit testing

In black-box unit testing, "[…] test cases are constructed based solely on the software's specification and not on how the software is implemented" [33]. "Black-box testing focuses on the functional requirements of the software" [46]. Black-box testing[1], traditionally a part of unit testing [25] [28], is a method used to test input and output values. When black-box unit testing, the tester does not consider the inner structure of the unit being tested.

Black-box test cases may be designed early in the development process, as soon as the unit's specification has been established. In other words, test cases may be designed before any code has been written.

There is a risk with black-box unit testing. If the specification is misinterpreted the misinterpretations may be propagated into the black-box test cases [33]. The misinterpretation problem, discussed in section 3.4.3, may be solved by including inspections.

Different methods that fall into the category of black-box unit testing will be discussed in the following two sections.

3.4.7 Equivalence partitioning and boundary value analysis

Equivalence partitioning and boundary value analysis are two methods that are closely related to each other. Pol et al. [45] state that boundary value analysis is a specialization of equivalence partitioning. Therefore, the two methods are described together.

Jacobson et al. [25] writes that "The technique of equivalence partitioning should be used in all testing." Equivalence partitioning is a test case selecting method that reduces the number of test cases needed to sufficiently test input, output and operation. Similar inputs, similar outputs and similar operations are grouped in equivalent partitions [41]. All values in an equivalent partition should cause an operation to behave in a similar way. As an example, for an operation that takes values between 0 and 10 as input, the values are divided into three partitions; one that equals 0 and below, one that lies between 0 and 10 and one that is equal to 10 an above (see Figure 3.7). Note that partition number two is called a valid equivalence

[1] Also known as behavioral testing or functional testing

Figure 3.7: Equivalence partitioning example

partition and partitions one and three are called invalid equivalence partitions.

Instead of testing all possible input values, only one test case for each equivalent partition is required. Selecting more test cases from the same partition "barely increases the chance of finding defects" [45].

When reducing the number of test cases, the risk of eliminating tests that could reveal faults should be considered. Therefore, each proposed equivalent partition should be reviewed [41].

When deciding equivalent partitions, valid and invalid equivalence partitions are identified because both have a high probability of finding faults [45]. Patton [41] explains that an equivalence partition, separated from the valid and invalid cases, should always be created that handles default, empty, blank, null, zero or none conditions because the software usually handles them differently. Test-to-fail[1] should be performed after test-to-pass[2] and should include testing wrong, incorrect and garbage data [41].

Pressman [46] states that equivalence partitioning is a black-box testing method. But it may also be used in white-box testing, e.g. testing array ranges.

"If software can operate on the edge of its capabilities, it will almost certainly operate well under normal conditions" [41]. This quote explains the main idea of boundary value analysis. When identifying equivalence partitions the partitions are separated by boundaries. The boundaries constitute extreme values that should be tested. The relationship between the equivalence partitioning and boundary value analysis may be described as in Figure 3.8.

[1] Testing with invalid values, e.g. testing with invalid or illegal inputs.
[2] Testing with valid values.

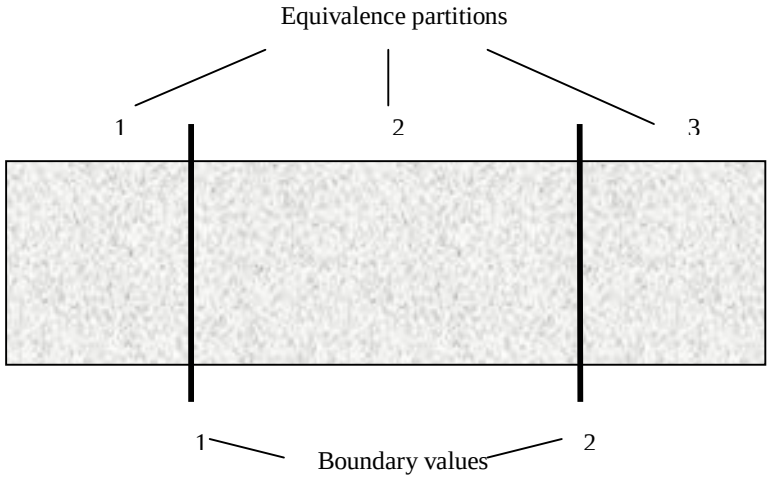


Figure 3.8: Equivalence partitioning and boundary value analysis relationship.

Hutcheson [21] states that boundary values are the most important test cases. The assumption of testing at boundaries is that if tests work at and around boundaries, tests also work between the boundaries (in the equivalent partitions). When choosing data to test in equivalence partitioning, more faults will be found if data is chosen from the boundaries [41]. Pol et al. [45] argues that a large number of faults occur in values located at and around boundaries. Testing should be performed on a boundary, and on both sides of the boundary [41].

Aside from the "normal" boundary conditions Patton [41] describes sub-boundary conditions or internal boundary conditions that are internal to the software and not apparent to the end user. The ASCII character table and binary numbers are two examples of sub-boundary conditions that need to be considered when determining partitions [41].

The conclusion is that equivalence partitioning and boundary value analysis are two test case selection methods that will be included in the UTM.

3.4.8 The all-pairs method

The all-pairs method is a method that involves testing several variables together, e.g. the arguments to an operation. This whole section, including examples, is based on the all-pair approach described by Kaner et al. [26].

An example will be used to describe the all-pairs method. Suppose an operation takes three variables as arguments, V1, V2 and V3, and that each variable has 100 possible values. By using equivalence partitioning and boundary analysis the number of values that need to be tested may be reduced. Assume that three values, A, B and C need to be tested for V1, two

values, X and Y for V2 and two values, 0 and 1 for V3. If all combinations of the values should be tested, that would require 3 × 2 × 2 = 12 test cases. Kaner et al. recommend that the number of test cases is further reduced by using the all-pairs method, which requires that the test cases should include all possible pairs of values for every combination of variables. The test cases needed for including all pairs may be identified using an all-pairs matrix, see Table 3.2. The all-pairs matrix is constructed by first labeling the columns with the variable

V1	**V2**	**V3**
A	X	0
A	Y	1
B	X	1
B	Y	0
C	X	0
C	Y	1

Table 3.2: All-pairs matrix

names, starting with the variable with the highest number of possible values, in this case V1, to the left, and continuing to the right in descending order of possible values. For the V1 column, groups of rows are constructed. The number of groups of rows is the same as the number of possible values for V1 (three), and each group contains one of the possible values of V1, see Table 3.2. The number of rows in each group in the V1 column is the number of possible values for the second leftmost column in the matrix, which are two for V2. When the V1 column has been filled, the V2 column is filled in a way so that each V2 value is paired with each V1 value at least once. Then finally, the V3 column is filled, making sure each possible V3 value is paired with each of the possible V2 values at least once, which requires four rows. The remaining two rows may be filled out randomly for the V3 column. The resulting matrix contains six rows, representing six test cases, which reduces the number of test cases by half. Using the all-pairs method may reduce the number of test cases with much more than half. There may however be important combinations of values for the variables that are not covered by the all-pairs method, so additional test cases should then be added.

The example used is simple. When dealing with more variables and a higher number of possible values it may be more difficult to fill out the matrix. It is however possible to fill the matrix by following the general outline described in this section. The all-pairs method will be included in the UTM.

3.4.9 White-box unit testing

Just like black-box testing, white-box testing is traditionally a part of unit testing [25] [28]. White-box testing[1] is in a sense the opposite of black-box testing. The unit to be tested is seen as a "white box", meaning that the internal structure of the unit is taken into consideration. Code constructs that control the path taken through the code, for example conditional statements and loops should be considered when designing white-box test cases [10].

Since exhaustive black-box testing often is a practical impossibility due to the large number of possible combinations of inputs, white-box testing should also be used so that combinations of input may be chosen that test the different paths through the code [10] [16]. In comparison with black-box testing, white-box testing has the advantage of improved code coverage [33], which means that more statements, branches and paths of the code have been tested. Pressman [46] states that one reason for white-box testing is that it is more probable that faults lie in paths in the code that are rarely executed. These paths may often be covered only through white-box testing. The main disadvantage with white-box testing according to McGregor and Sykes [33] is that "if the programmer did not implement the full specification then that part of the functionality will not be tested." Testing that the unit has implemented the main functionalities is however handled by black-box testing.

Different methods that fall into the category of white-box unit testing will be discussed in the following four sections, 3.4.10 to 3.4.13.

3.4.10 Path testing

The purpose of path testing is to exercise the execution paths throughout a program. A problem with path testing is that even small-sized program units may have a large number of paths. It is simply not possible to test every path in a unit. As an example, consider an operation that contains two nested loops where each loop may run one to ten times. If the unnested loop runs only one loop, the nested loop may be run in ten different ways. If the unnested loop runs two loops, the nested loop may run in 10^2 ways and so on. Totally, the number of different paths through the nested loop is:

$10 + 10^2 + 10^3 + 10^4 + 10^5 + 10^6 + 10^7 + 10^8 + 10^9 + 10^{10} = 11111111110$

Instead of trying to test all paths a subset of all paths with high probability of finding faults should be identified. Basis path testing is a test method used to test a subset of all possible

[1] Also known as glass-box, clear-box or structural testing

paths. Basis path testing makes use of the cyclomatic complexity[1] measure and flow graphs[2] to identify all independent paths. The number of test cases used to exercise program paths should be at least as many as the value of the cyclomatic complexity metric [50], i.e. the number of independent paths. However, there are difficulties with identifying independent paths and using the cyclomatic complexity measure. Consider the following flow graph:

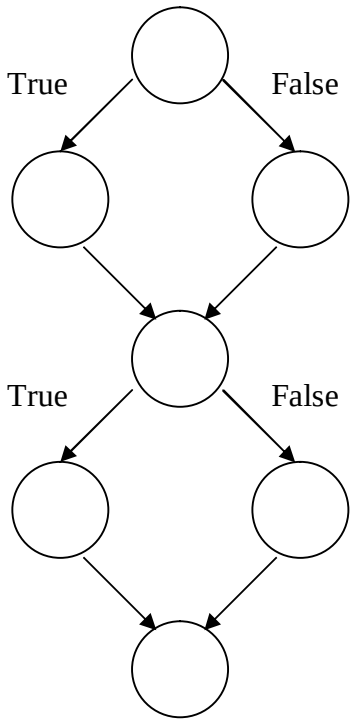


Figure 3.9: Independent paths-problematic flow graph

The cyclomatic complexity value for the graph in Figure 3.9 is:

$$C(G) = 8 - 7 + 2 = 3$$

Therefore, there exist three independent paths in the graph. Pressman [46] describes an independent path as a path that runs through a new edge within the flow graph. By Pressman's definition, there exist only two independent paths in Figure 3.9 in contradiction to the cyclomatic complexity value. Actually, there exists one more independent path, but the path does not run through a new edge. If one path follows only the true conditions, and one path follows only the false conditions, a third path needs to be exercised in order to prove that the two decision nodes do not always execute the two previously described paths. Therefore, the path that runs through the first true branch and then the false branch or vice versa should be run to show the independency between the two decision nodes. Whenever two or more

[1] Cyclomatic complexity is described in Appendix D.
[2] Flow graphs are described in Appendix E.

non-nested decision nodes follow sequentially, the problem described with Figure 3.9 occurs. Therefore, it is important to identify the extra independent paths that do not introduce new edges but exercises the independency between the decision nodes.

If every independent path has been exercised it is certain that [50]:

1. “every statement in the method has been executed at least once”
2. “every branch has been exercised for true and false conditions”

Branch testing is a testing method which main purpose is to exercise every branch of a decision in a program. If basis path testing is performed, branch testing becomes superfluous.

Sommerville [50] explains that there exist testing tools called dynamic program analyzers. These testing tools may determine how many times each statement has been executed and also reveal untested sections of the program. Dynamic program analyzers should be used when performing path testing to determine which branches have been and have not been executed.

The conclusion is that basis path testing will be used in the UTM. Each independent path is recommended to have its own test case in order to exercise every statement at least once and exercise every branch. Tools that support automation of drawing flow graphs, automation of cyclomatic complexity calculation and identification of the statements and branches executed are recommended to use.

3.4.11 Data flow testing

Data flow testing is also considered a path testing method, as described in the previous section. Data flow testing focuses on locations of the definition, usage and termination of variables.

- *Definition:* The concrete state of a variable is changed, e.g. an initialization operation or a constructor.
- *Usage:* A variable is used either for computing or in a predicate.
- *Termination:* The variable exits its original scope, e.g. deallocated.

For each variable, the definition, usage and termination of the variable throughout an operation are identified. A data flow path is a definition-usage-termination path. Test cases for each data flow path should be designed to achieve full coverage.

Pressman [46] states that data flow testing is effective for error protection, but selecting test paths and measuring test coverage is difficult. Binder [11] explains that if several paths exist between a definition and use of a variable, only one need to be taken in order to achieve effective testing. Also, according to Binder [11], if a fault is located on a data flow path and that path is never executed, testing will probably never reveal that fault. Therefore, even though it is difficult to select test paths in data flow testing and to measure test coverage, data flow testing is important to perform in a testing process in order to find faults associated with data flow paths. Data flow testing will be included in the UTM.

3.4.12 Multiple condition testing

Branch testing does not take in consideration whether a decision is based on a compound condition or not, only the true or false path. Multiple condition testing is concerned with exercising, at least once, all true and false combinations of simple conditions[1] [11]. Possible types of components in a multiple condition are Boolean operators, Boolean variables, parentheses, relational operators and arithmetic expressions [46]. Combinations of component truth values to be tested in a multiple condition may be put into a truth table, which gives a structured approach to cover all possible combinations. Multiple condition testing will be included in the UTM.

3.4.13 Loop testing

"Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs." [46]

- *Simple loops.* A single loop.
- *Concatenated loops.* Two or more loops in sequence on the same control path.
- *Nested loops.* Loops contained within loops.
- *Unstructured loops.* Loops that contain very complex flow paths.

Loop testing will be included in the UTM. Details on how these four different loop constructs should be tested are described in Appendix I.

[1] "A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT ("¬") operator." [46]

3.4.14 Testing exceptions

Exceptions and exception handling are terms which are often used interchangeably. “Exception handling is a programming language construct or computer hardware mechanism designed to handle runtime errors or other problems (exceptions) which occur during the execution of a computer program.” [17]

At a class-testing (unit-testing) level every exception should have at least one test case [33]. If a class under test throws exceptions each test should be designed to catch the exceptions. Exceptions not caught may not be evaluated [11].

Binder [11] states that failures with respect to an exception may occur in four ways:

- An exception is thrown when the exception should not be thrown.
- An incorrect exception is thrown.
- An exception is not thrown when the exception should be thrown.
- An exception thrown by a class that the class that is tested uses is not caught.

Test cases must catch all exceptions to reveal the faults mentioned in the list above [11].

A try block may be used in the implemented test case to test a specific exception and verify that it is the correct exception [33]. A test case should both verify whether the occurrence as well as the absence of exceptions is a correct response. Unexpected exceptions may be caught using a “catchall”[1] construct, which is important to use if the exceptions are to be detected [11].

If postcondition clauses are a part of the class specification then exception testing becomes a normal part of unit testing, since descriptions of exceptions thrown by an operation should be included in the postcondition clause [33].

The conclusion is that each exception should have at least one test case. The implemented test case should include a try block that verifies if the exception being caught is the correct exception and if the exception is thrown at the right time. To check if unexpected exceptions are thrown, the test case implementation should include a “catchall” construct. These guidelines for testing exceptions will be included in the UTM.

[1] An example of a catchall construct is the use of ’catch’ in Visual Basic .NET. If catch is used without ’as’ (catch…as) or ’when’ (catch…when) any kind of exception may be caught.

3.4.15 Unit test methods for object-oriented code

The methods discussed in the following sections, sections 3.4.16 to 3.4.20, are methods that are specific for unit testing object-oriented code, which are represented by the shaded area in Figure 3.10 (derived from Figure 3.1 in Section 3.1).

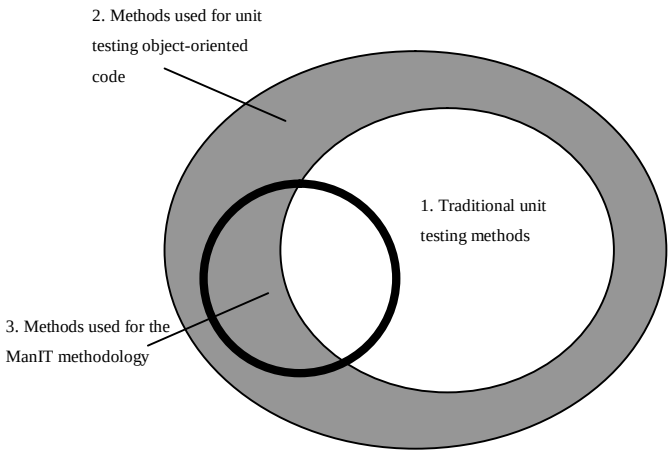


Figure 3.10: Test methods specific for object-oriented code

3.4.16 State-based unit testing

State-based unit testing is based on the concept of state. A state of an object is a subset of "the set of all possible combinations of instance variable values" [11]. The possible states of an object and the transitions between the states are considered when designing test cases. The states may reflect either the external behavior of the object (i.e. a black-box approach) or the internal representation and algorithms of the object (i.e. a white-box approach) [13]. In a certain state, some operations may be valid, and should be tested so that the correct behavior, e.g. transition of state, is observed. Some operations may not be valid in a certain state, and should be tested so that, for instance, an appropriate exception is thrown, or the state remains the same. Also, sequences of operations should be tested [11].

The interactions, dependencies and constraints on operation sequences of an object are called behavior [11]. According to Binder [11] state-based testing is an effective, fundamental technique for testing complex behavior, and thus "object-oriented software is well suited to state-based testing". Two of the problems with state-based testing (that are relevant for unit testing) are [54]:

1. "It may take a very lengthy sequence of operations to get an object into some desired state."

2. “State based testing may not be useful if the class is designed to accept any possible sequence of method calls.”

The first problem is a matter of time (and therefore cost). If enough time is available, there is no problem with having to use a long sequence of operations. Concerning the second problem, Binder [11] states that for objects that have few constraints on operation sequences, state-based testing is inappropriate.

The conclusion is that state-based unit testing is useful when distinct states of an object may be identified and a number of constraints on the possible operation sequences exist. State-based unit testing will be included in the UTM.

3.4.17 Selecting test cases for state-based unit testing

First, the possible states of an object must be established. It is not always possible to have each combination of possible member values of an object as an individual state [25]. Instead, sets should be identified for combinations of member values for which the code of the class under test (CUT, the class to be tested) should behave in a similar way [25]. A state transition diagram may be used to visualize the states and transitions of an object [25] (The UML statechart is for instance recommended by Binder [11]). For the following discussion, a stack with only push and pop operations, based on [54], will be used as an example. Figure 3.11 shows a state transition diagram for the stack. The two states for the stack are ‘empty’ and ‘loaded’. The predicate expressions associated with the pop operations are called guards [11]. The guard predicate must be evaluated to true for the transition associated with the guard to take place. Jacobson [25]

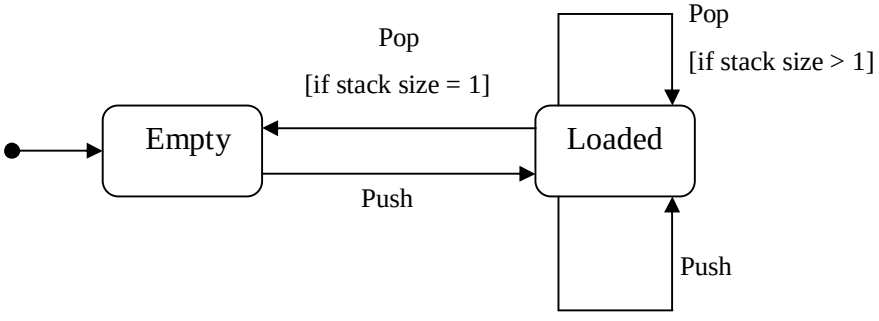


Figure 3.11: State transition diagram for a stack

states that when a state transition diagram is used, test cases should be chosen so that every state is visited at least once and every transition is traversed at least once. Binder [11] however, argues that each event should be tested in each possible state. Events are:

- Operation calls to the CUT.
- Responses from other services that the CUT uses.
- Interrupts/exceptions that the CUT must respond to.

In the stack example the events are ‘create’, ‘push’ and ‘pop’. McGregor and Sykes [33] claim that boundary values must also be considered, both for the states and for the guards. Boundary values for states are determined based on the range of member values associated with that state [33]. In the stack example the boundary between the two states lies between an empty stack and a stack size of one. The boundary values for the guards in the stack example are the same as the state boundary values.

To help determine state-based test cases a response matrix may be used [11]. A response matrix shows all combinations of events and states, and also includes the guards. Table 3.3 shows a response matrix for the stack example. The rows of Table 3.3 correspond to events

Events and guards		**Accepting state/Expected response**	
		Empty	**Loaded**
Create		Stack created size = 0	
Pop	*size = 1*		
	DC	popEmptyException thrown	
	T		State changed to Empty size = 0
	F		size = size - 1
Push		State changed to Loaded size = 1	size = size + 1
Key: T = true, F = false, DC = don’t care			

Table 3.3: Response matrix for Figure 3.11

and guards. Only ‘pop’ has guards in the stack example. The columns to the right in the response matrix in Table 3.3 represent the states. In the intersections of the events/guards and the states, the response expected when the event occurs in the state is noted. For instance, when the event ‘Pop’ occurs in the ‘Loaded’ state and the size is equal to one, the state is changed to the ‘Empty’ state and the size is set to zero. The ‘X’ is used when events are not accepted in a state, or if the event is already covered in that state. For example, the ‘Pop’

event with the size guard as DC (don't care) is marked with an X in the 'Loaded' state, since the two possible cases for the size guard, true or false, are already covered in the 'Loaded' state.

The most important cell in the response matrix in Table 3.3 is the cell representing a 'Pop' event in the 'Empty' state, not caring about the guard. It should not be possible to pop an empty stack, but the stack should respond to the event (unless programming with strong contracts [38], which means that the client of the stack is responsible for using the stack correctly). In this example, an exception is thrown, popEmptyException. If only the state transition diagram is used to derive test cases, the test case for the popEmptyException may be overlooked. Binder [11] calls these types of responses that are not visible in the state transition diagram implicit responses.

The conclusion is that state transition diagrams and response matrices are useful for selecting test cases for state-based unit testing. The UTM will recommend the use of both state transition diagrams and response matrices.

3.4.18 Inheritance

According to Lindegren [29], inheritance complicates unit testing in object-oriented systems. In an inheritance tree, there are dependencies between the classes, and the classes should not be unit tested independently [29].

Consider first if a class is an abstract class. McGregor and Sykes [33] state that they prefer to unit test abstract classes as a part of testing the first concrete descendent subclass. The concrete subclass used should not override any of the abstract class' operations [33]. Test cases that test the implemented parts of the abstract class should be added to the tests of the concrete subclass [33].

Suppose that class B inherits class A (see Figure 3.12). Also assume that the inheritance

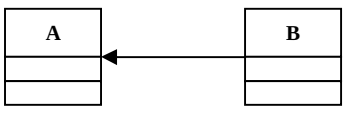


Figure 3.12: Class B inherits class A

follows Liskovs substitution principle[1] (LSP), which according to McGregor and Sykes, is "good object-oriented design" [33]. Using inheritance in accordance with LSP implies that the test cases for a class are valid also for a subclass of that class [33]. If inheritance does not follow LSP the superclass test cases are of limited use in the subclass [11] and although a number of superclass test cases may be reused for the subclass, many test cases probably need to be redesigned.

Consider the following three ways of refinement in subclass B (Figure 3.12) and the corresponding affect on testing (based on McGregor and Sykes [33] and Binder [11]):

1. Adding a new operation in B – A new operation does not directly affect existing, inherited operations. New test cases have to be designed for the new operation. New test cases must also consider how the new operation interacts with the inherited operations, e.g. sequences of operation calls.
2. In B, changing the specification of an operation inherited from A – New black-box test cases need to be designed for the refined operation, e.g. to meet weakened preconditions or strengthened postconditions. The test cases from A for the operation need to be rerun, and may need adjustment in B to consider strengthened postconditions. For any inherited operation that is affected indirectly, the test cases from A need to be rerun.
3. In B, overriding an operation inherited from A – Black-box test cases from A for the overridden operation may be reused. Other test cases from A for the operation need to be reviewed and revised in B, and new test cases for the overridden operation may be needed. For any inherited operation that is affected indirectly, the test cases from A need to be rerun.

To analyze which of the test cases from A that need to be rerun and which test cases do not need to be rerun is "tricky" [33]. McGregor and Sykes [33] states that "in practice, it is usually easier and more reliable to just rerun all test cases.", and Binder [11] agrees. Lindegren [29] also argues that all superclass test cases should be rerun when unit testing subclasses, preferably with aid from automated test tools.

[1] One way of stating the Liskov substitution principle is: "if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program [30]" In the context of section 3.4.18 the types are classes.

Consider the following two different methods to achieve reuse of the test cases of a superclass in a subclass (based on the test driver design patterns described by Binder [11]):

1. Implement the test cases as operations in the CUT – The test cases will be inherited just as any other operation and may easily be reused.
2. Use a parallel test class hierarchy – A test class is a class that contains the test cases for a CUT. If class B is a subclass to class A, then the test class for class B should be a subclass to the test class of class A (see Figure 3.13). B thus inherits the test cases from A.

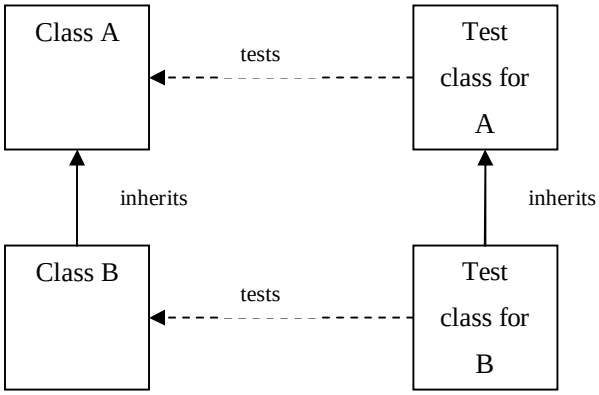


Figure 3.13: Parallel test class hierarchy

An extensive number of both advantages and disadvantages with both approaches are described by Binder [11]. Some advantages of method number one are:

- No extra test class hierarchy to maintain.
- Information hiding is not a problem; the test cases may access the private members of the CUT directly.

Some disadvantages of method number one are:

- Faults in the test cases may induce unwanted side effects.
- The test code may be misused (for instance test code to set state).

Some advantages of method number two are:

- The encapsulation of the CUT is respected.

- The test cases are organized “in direct correspondence to the structure of the classes under test” [11].

Some disadvantages of method number two are:

- There is an extra cost of maintaining an additional class hierarchy.
- Private and protected methods may not be exercised directly.

The list of disadvantages provided by Binder [11] is longer in the case of method number one. Most of the disadvantages of method number two are due to information hiding and may be compensated for (described in section 3.4.20). McGregor and Sykes [33] claim that the structure visualized in Figure 3.13, corresponding to method two, “presents a clean organization and is easy to implement.”

The conclusion is:

- Abstract classes should be unit tested as a part of unit testing the first concrete descendent subclass. Preferably, a concrete subclass that overrides as few of the abstract class’ operations as possible should be used.
- The test cases for a class hierarchy should be organized as a parallel test class hierarchy.
- In inheritance hierarchies that follow Liskovs substitution principle all the inherited superclass test cases should in a subclass
    1. not be rerun, if only new operations are added in the subclass.
    2. be rerun, and possibly be modified if superclass operations are refined in the subclass.
- In inheritance hierarchies that do not follow Liskovs substitution principle a number of superclass test cases may be reused, but many test cases probably need to be redesigned.

The guidelines for testing inheritance hierarchies in this section will be included in the UTM.

3.4.19 Polymorphism

Dynamic bindings and polymorphism may complicate unit testing. “In the case of object oriented programming each possible binding of a polymorphic class requires a separate set of test cases. The problem for unit testing is to find all such bindings – after all the exact binding used in a particular object instance may only be known at run-time.” [54] To handle all possible bindings for a variable, all classes that may be dynamically bound to the variable should be considered and test cases should be designed that cover the bindings. Polymorphism testing will be included in the UTM.

3.4.20 Information hiding

The use of information hiding in object-oriented programming may obstruct unit testing. Information hiding is the idea of only showing a class’ interface to clients of the class and keeping the implementation hidden. Private operations and members may normally not be accessed from outside the class. If the test cases are implemented in another class than the CUT (class under test), the problem is how to unit test the private operations and members of the CUT.

As described in Section 3.4.18 a parallel test class hierarchy will be used for the test cases in the UTM. As in the parallel test class hierarchy method, collecting test cases for a class in a separate test class will also be used for single classes that are not part of a hierarchy, to achieve consistency in the UTM.

The state of private members may often be both observed and set using public operations of the CUT, but may require long sequences of operation calls [11]. An advantage of observing and setting a private member directly through private access is that it may be more effective [11]. A disadvantage is that the test class is more tightly coupled to the implementation of the CUT [11].

For the test class to gain access to the private parts of a CUT, language specific mechanisms may be used [11]. For example:

- In the .NET framework class library there is a namespace called System.Reflection which contains functionality to access private parts of other classes during runtime by using a mechanism called reflection, which is the ability to get information of an object during runtime, and even invoke (private) operations [12] [15].
- In the .NET supported languages C++ and Visual Basic .NET a class may be declared as a “friend” by another class. By letting the CUT declare that the test

class is a friend, the test class is able to access the private parts of the CUT. This solves the problem of unit testing private operations, and helps to make it easy to set and get concrete states of objects [11].

Although using friend makes it necessary to include a line of code in the CUT, using a single line is easier than using the more complex syntax [12] [15] of the System.Reflection library. Also, accessing private methods using System.Reflection requires permissions to be set [56]. McGregor and Sykes [33] recommend the use of 'friend'. A final argument for primarily using the keyword 'friend' is that Visual Basic .NET, which is the preferred language at ManIT, supports 'friend'.

One way to avoid the problem that information hiding imposes on unit testing is to use assertions in the private parts of the code of the CUT [11]. Assertions are "boolean expressions that defines necessary conditions for correct execution" [11]. The assertion code is written in the implementation of the CUT, and may be executed during runtime [11]. Assertions are convenient to use together with contracts, consisting of pre- and postconditions and invariants [6] [11]. According to Binder [11], assertions are not always practical. Contracts make it easier to use assertions [11], but the existence of contracts may not be guaranteed in the situations where the UTM will be used. The use of assertions will also split up the unit testing, with test cases in the test classes and assertions in the implementation of the CUT. The use of a language specific technique such as friend makes it possible to have all tests together in the test classes, which provides a consistent organization of the tests.

The conclusion is that:

- If suitable, the public interface of the CUT should be used by the test class for observing and setting private members.
- For accessing private operations and possibly private members, language specific techniques should be used. If the keyword "friend" exists in the language used, it is preferred. If "friend" is not available, the .NET System.Reflection library should be used.

The methods for testing non-public members and operations, described in this section, will be included in the UTM.

3.4.21 Order of testing within a unit

This section deals with how executing test cases of a unit in a particular order may simplify the unit testing. The following is recommended:

- Test constructors, getters and setters[1] first, since other operations depend on them [33]. Also, the unit test drivers themselves may use constructors and setters to put the object under test in a desired state, and may then use getters to check if the result of the test case is correct [33].
- Test single operations before testing the interaction of several operations [26]. That is, before testing sequences of operations as with state-based unit testing, make sure all the single operations work as intended.
- Test superclasses before their subclasses [40]. The test cases for the superclass may then be reused for the subclasses.
- Test 'helper' operations (often private operations) before the operations that use the 'helper' operations. Then when testing an operation that uses a 'helper' operation, the 'helper' operation will work as intended.

These guidelines for the order of testing within a unit will be included in the UTM.

3.4.22 Documentation of unit test cases

SEMM [62] states the requirements for test case documentation for test cases designed during the formal testing activity. These requirements are not mandatory for unit test cases that are to be executed during the implementation activity [62], as is the case for the UTM. The requirements that are applicable for unit testing will however be included in the UTM to achieve conformity. The information sections provided for each test case in the test case documentation should be few, since there will be several unit test cases to write and too much administrative procedures should be avoided. The following information should be provided for a unit test case:

- *Test case ID* – A unique ID among all the test cases of the unit [41] [46].
- *Test item* – The class, classes, operation or/and operations under test.

[1] A 'getter' is an operation that returns the value of a member of an object. A 'setter' is an operation that allows the client of an object so set the value of a member of the object.

- *Description* – A short description of the test case, for example: “Pop the stack with only one value in the stack.”
- *Input*
    1. **Setup** – The necessary setup needed to perform the test, for example: “Create a stack object and push the value 1 onto the stack.”
    2. **Activities** – The activities needed to perform the test, for example: “Call the pop operation of the stack object.”
- *Result state* – The results expected from the test, for example: “The value ‘1’ is returned, the stack is in the ‘empty’¨ state and stackSize = 0”.
- *Additional information* – Any other information that will aid in understanding or implementing the test case, for instance special hardware/software required or the relationship of the test case to specific unit requirements/design aspects.

The ‘Input’ and ‘Result state’ sections are included in the ‘Description’ section in SEMM, but for the UTM individual sections have been used for the ‘Input’ and ‘Result state’ as described by McGregor and Sykes [39]. Individual sections make it more clear what information to provide in each section.

Since a ManIT requirement is that the unit test should be automated, no detailed step-by-step procedures of how to perform the unit test cases, as described by SEMM and Perry [42], are necessary.

Patton [41] strongly recommends using spreadsheets, or a specific database for handling test cases, to organize the test cases instead of using paper documents. A spreadsheet stored in a single place or a database makes it possible to have only one version of the documentation. Paper documentation is hard to manage and keep up to date, especially if the design changes frequently [21].

According to SEMM [62], the documentation of the unit test cases should be included in, or referenced by, the test plan developed during the test planning activity.

The guidelines for how the unit test cases should be documented, described in this section, will be included in the UTM.

3.4.23 Unit test drivers

A test driver is code that executes one or more test cases. Test drivers may be designed in different ways, but implementing the test drivers in separate test classes is the recommended way by McGregor and Sykes [33]. This is consistent with using a parallel test class hierarchy

as described in section 3.4.18 concerning inheritance. The individual test cases for a unit are implemented as operations in the test class for the unit, as in the “symmetric driver” test pattern described by Binder [11].

To make it possible for a subclass to use the inherited test cases of a superclass, the CUT object should be passed as an argument to the operations (test cases) in a test class instead of being hard coded in the implementation of the test case operations. Then a test class for a subclass may provide objects of the subclass type as arguments to the test case operations inherited from the superclass test class. The superclass may for instance provide an operation that takes a CUT object and then calls all the test case operations, as showed in the Visual Basic similar pseudo code in Figure 3.14.

```
Public Class TestSuperclass
   Sub runAllTestCases(testObject As Superclass)
      testCase01(testObject)
      testCase02(testObject)
   End Sub
   Sub testCase01(testObject As Superclass)
      ...
      End Sub
   Sub testCase02(testObject As Superclass)
      …
   End Sub
End Class
```

```
Public Class TestSubclass
      Inherits TestSuperclass
   Sub runAllTestCases()
      Dim testObejct As New Subclass
      MyBase.runAllTestCases(testObject)
   End Sub
End Class
```

Figure 3.14: Pseudo code for inheriting superclass test cases

The execution of the drivers may be automated with the aid of computer aided software testing tools [41].

Using unit test case drivers will be recommended in the UTM.

3.4.24 Stubs

A stub is a temporary, minimal implementation of a class [11]. The interface of a stub should be identical to the interface of the class the stub is replacing, it is only the implementation that should be different [11] [13]. Both classes and operations within the CUT may be replaced by stubs [11]. Binder [11] claims that stubs should be avoided if possible. If the development follows a bottom-up approach, where units being used by other units are developed first, few stubs will be needed [11]. According to Binder [11] stubs may be useful in the following ways:

- Classes used by the CUT that are not yet developed may be replaced by a stub.
- When classes depend on each other in a cyclic way, a stub may replace one of the classes so that the other ones may be unit tested independently.
- When exceptions thrown by another class are to be handled by the CUT, the other class may be replaced by a stub that generates exceptions in a controlled way.
- Messages sent from the CUT may be checked in a stub.

Advantages of stubs are that it is easier to control and observe a stub than the real implementation of the class that the stub replaces [11]. For example, if the CUT sends a message to another class for which it is difficult to observe the results of the message, the other class may be replaced by a stub constructed in such a way that the effect of the message may be easily observed.

Two of the disadvantages of using stubs are [11]:

- "For small [class] methods, the time and effort to develop a stub are often about the same as they are for the actual [class] method."
- The behavior of the fully implemented object may differ from the stub. Thus the result of running the test cases with the stub may not be valid for the final implementation of the object.

According to Caspersen et al. [13] describing guidelines for when to stub is difficult. Classes that several other classes depend on may be useful to stub [13]. Otherwise, the dependencies should be tested during integration testing, using the final implementation of the classes [13]. The question of whether to use a stub and test a class separately or to wait and test the classes together is another example of the vague border between unit testing and integration testing in object-oriented systems.

Kruchten [28] states that stubs developed should also be unit tested. To have test cases for the stub may also be useful if the stub is further developed into the final implementation, which is recommended [11] [13].

Using stubs will be recommended in the UTM. Stubs will be recommended to use when appropriate according to the four point bullet list in this section that describes when stubs may be useful, as recommended by Binder [11].

3.4.25 Prioritizing unit tests

Resources such as time, money or staff may not be available to test all units of a software system adequately. Then, risks have to be identified [45] and the units with the most critical risks should be prioritized for testing [29]. According to Hutcheson [21] “risk analysis demonstrates the use of a best-practice approach, and unifies the teams view with respect to scope and priorities.” In addition to priorities based on risk, Sommerville [50] states that “it is more important to test the parts of the system that are commonly used rather than parts which are only rarely exercised”.

McGregor and Sykes [33] argue that “for each class, we must decide whether to test it independently as a unit or in some way as a component of a larger part of the system. We base that decision on the following factors:

- The role of the class in the system, especially the degree of risk associated with it.
- The complexity of the class measured in terms of the number of states, operations, and associations with other classes.
- The amount of effort associated with developing a test driver for the class.”

The choice between testing a class as a unit during unit testing or as part of a larger component again illustrates that unit testing and integration testing are overlapping concepts for object-oriented systems.

Asides from prioritizing among units, prioritizing among unit testing methods may also be necessary. Bell et al. [10] claim that “different test techniques tend to discover different errors. Therefore the more techniques that are employed the better, provided there is adequate time and money.” It is agreed upon that both black- and white-box methods should be used when unit testing [10] [16] [33] [46]. However, if adequate time and money are not available, black-box testing is the more important approach [16] [33] [40], and should therefore be prioritized.

How the testing should be prioritized will be described in the UTM.

3.4.26 Criteria for ending unit testing

Patton [41] states that it is impossible to test a program completely due to four key reasons:

- “The number of possible inputs is very large.”
- “The number of possible outputs is very large.”

- “The number of paths through the software is very large.”
- “The software specification is subjective.”

By multiplying these “very large” possibilities a number of test conditions are derived that are too large to attempt [41].

According to Pressman [46] testing is never completed but there exist guidance to determine when testing is done in a testing process. If testing is viewed as an activity that is not only performed during a project but performed by the customer after product release then testing is never done. Another view of when testing is done is when resources like time and money runs out [46]. Eklund and Fernlund [16] write that the question when testing is finished is mainly an economic issue. McGregor and Sykes [33] explain that to determine how much testing should be done on a piece of software the characteristics of the software need to be considered, e.g. if the software is life-critical or non-critical, and the lifetime of the software. Another consideration is the balance between the cost of uncovering a fault and the increased quality of the product. How much testing needs to be spent testing a class may be determined if the class is identified as a reusable asset (more testing) or if the class will be used as part of a prototype (less testing) [33].

A continuum may be used in a project to indicate how much unit testing should be performed within the project (see Figure 3.15). Exhaustive testing runs every possible test case while no testing indicates no test cases or only a few [33].

No testing Exhaustive testing

Figure 3.15: Continuum for how much unit testing may be done [33]

Pressman [46] writes that using different metrics may indicate when to stop testing. Meaningful guidelines, which answer the question “When are we done testing?” may be developed from collecting different metrics throughout the test process [46]. McGregor and Sykes [33] argue that some measure of adequacy is needed to give a high level of confidence in the quality of test cases. A measure category that is commonly used when measuring how much testing has been performed is test coverage. Test coverage is a measure of how completely a set of test cases exercises the capabilities of a piece of software [33]. Test

coverage measures may roughly be categorized as coverage of code and coverage of requirements. Unit testing is mainly involved with code coverage.

Beizer [9] describes through Pfleeger [44] Figure 3.16 that shows different test coverage strategies and their relative strengths. The strongest coverage strategy is “all paths” while the weakest is “statement”. In general, stronger coverage strategies require more test cases but will cover the code more thoroughly, and therefore increase the reliability of the software.

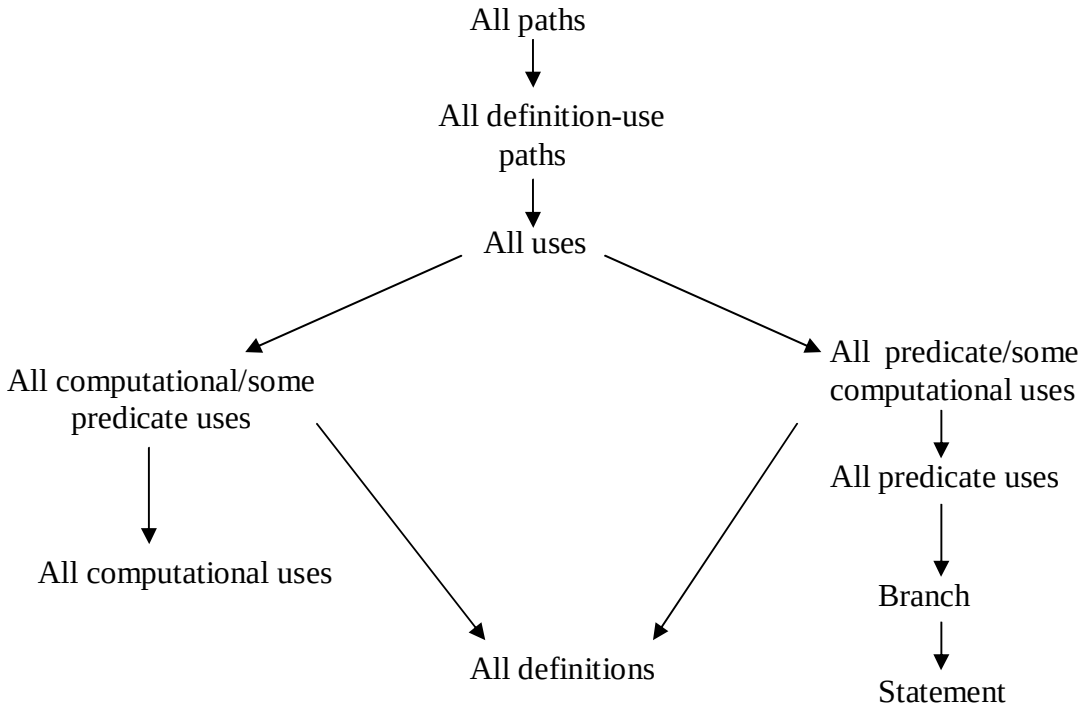


Figure 3.16: Relative strengths of test coverage strategies [9]

Before describing the different coverage strategies, the following definitions are needed [37] [48]:

- **Definition** – Assigning a value to a variable.
- **Def-clear path** - A path containing no redefinitions of a variable x along the path is called a def-clear path with respect to x.
- **P-use** – Short for “predicate use”. A p-use means that a variable is used in a predicate. Each different branch of a predicate results in a p-use for a variable used in the predicate.

- **C-use** – Short for “computational use”. A c-use means that a variable is used for computational purpose.
- **Use** – a p-use or a c-use.
- **Loop-free path** – A path in a graph with no repeated nodes.

Observe that all uses of a variable may not be reachable from all definitions of the same variable.

Now, the coverage strategies are defined [37]:

- **All definitions**: For each definition, at least one def-clear path from the definition to a reachable use should be executed.
- **All computational uses**: At least one def-clear path from each definition to each reachable c-use should be executed.
- **All computational/some predicate uses**: At least one def-clear path from each definition to each reachable c-use should be executed. If a definition does not reach a c-use, then to at least one reachable p-use.
- **Statement**: Every statement in the unit should be executed at least once.
- **Branch**: Every branch in a flow graph of the unit is included in the path of a test case for the unit.
- **All predicate uses**: At least one def-clear path from each definition to each reachable p-use should be executed.
- **All predicate uses/some computational uses**: At least one def-clear path from each definition to each reachable p-use should be executed. If a definition does not reach a p-use, then to at least one reachable c-use.
- **All uses**: At least one def-clear path from each definition to each reachable use should be executed.
- **All definition-use paths**: All feasible loop-free def-clear paths from each definition to each reachable use should be executed.
- **All paths**: All possible paths through the code should be executed.

The basis path testing method achieves branch coverage. Data flow testing covers definition-use paths, but not automatically *all* definition-use paths. For the other testing

methods it is not possible to determine a specific level of coverage from Figure 3.16 that is always reached.

There are also other coverage strategies that do not fit in the model illustrated in Figure 3.16, but that have relevance for the UTM:

- Multiple condition coverage requires that all possible combinations of the expressions in a predicate are exercised. The multiple condition testing method achieves multiple condition coverage.
- Two additional coverage strategies that are important when performing state-based testing are transition coverage and state coverage. Transition coverage means that every transition between states has been executed at least once during testing. State coverage means that every state has been visited at least once during testing. When using a response matrix for finding state-based test cases, as described in section 3.4.16, both transition and state coverage is reached, and also additional implicit responses are tested.

Software tools that automatically measure coverage of the code being tested are commercially available [33]. These tools show which parts of the code have been executed and therefore also indicates which parts of the code have not been executed.

According to Binder [11], both IEEE and IBM require statement coverage as a minimum coverage when testing code. Miller et al. [35] through Pedrycz and Peters [43] state, when describing guidelines for using coverage testing strategies, that the minimum coverage for branch testing is 85 %.

Additional criteria for ending the unit testing may be established, besides coverage criteria. These additional criteria may for instance involve the number of faults found or the duration of the unit testing. For example:

- Testing should be performed until no fault has been found for two whole days.
- Testing should be performed until a minimum of X faults have been found, or two weeks have elapsed without reaching X faults.

The conclusion is that every project should, during the test planning activity, establish goals for the unit testing. The goals for the level of testing for each unit should be documented

in the test plan. These goals should include test coverage measures that should be reached depending on the characteristics of the software to be developed. The test coverage measures may be different for different units. As a recommendation, every statement should have been executed at least one time and the branch coverage should be between 85% to 100%. Path, predicate, computational and definition-use coverage measures should be included in a project to a certain degree, depending on the software. It is recommended that software tools are used to automate the process of calculating which parts of the code have been covered. Additional goals beside test coverage measures may be established for the units.

How to determine criteria for ending unit testing will be described in the UTM according to the conclusions of this section.

3.4.27 Redundant test cases

Redundant tests are not useful in testing. Therefore redundant test cases should be avoided. But identifying redundant test cases may be difficult. As an example, consider black-box and white-box testing methods. If a black-box test has been performed there is a possibility that a number of paths throughout the unit being tested have been covered, and therefore these paths do not need to be white-box tested. It is up to the tester to identify and avoid redundant test cases.

3.4.28 Reporting unit test results

3.4.28.1 Fault reports

One of the reasons for reporting faults is that when the tester is not the person fixing the faults, the fault report is used by the person who actually fixes the faults [26]. Considering unit testing, it may be the unit implementer who both tests the unit and fixes the faults found. There are still other reasons for reporting faults:

- Information such as number of faults found may be used to determine the effect of the unit testing. The information may be used to convince management that the testing was worthwhile [21] [33].
- The fault reports may show that many faults were found in a certain unit, which may indicate that the unit is fault-prone and complex and requires further testing [42] [44].
- The fault reports may be used to derive metrics [44] used to improve the UTM [42].

Pol et al. [45] claim that fault reporting is not necessary for unit testing. The reason is, according to Pol et al., that the programmers

1. may correct the faults immediately when found, and thus do not lose time administering faults.
2. "avoid their programming defects becoming public knowledge".

Pol et al. make the assumption that the programmer of the unit is also the tester and the debugger of the unit. This assumption is probably valid at ManIT. The disadvantage pointed out by Pol et al. is that there is no insight in the quality of the unit testing. Since the UTM developed has not been used before, assessment of the quality of the methodology is important. Reporting the faults found when using the UTM (for instance in the case study described in Chapter 4) is necessary to be able to improve the methodology. Concerning point two about public knowledge of faults (defects), the programmer must be informed that faults are not reported to lay blame on the programmer [26]. To have found many faults in a unit may be positive, indicating that the unit was well tested.

Patton [41] states that "fault descriptions should include the minimal facts and details needed to describe the fault", and also claims that the descriptions should be clear and explicit. The fault report should for each fault contain (based on Perry [42], Kaner et al. [26] and Patton [41]):

- *Fault ID* – A unique ID among all the faults of the unit.
- *Date the fault was found* – May be used to gather metrics.
- *Situation of occurrence* – Information about the situation when the failure(s) occur(s). For example: "When the stack is in the empty state and the 'push' operation is called with the argument 0."
- *Effect of the fault* – Information about the effects of the fault (e.g. observed failures). For example: "An exception is thrown instead of 0 pushed on the stack."
- *Fault description* – For example: "Checking if the argument is >0 instead of >=0 in the 'push' operation."
- *Impact of the fault* – discussed later in this section.
- *Type of fault* – discussed later in this section.

- *How the fault was found* – For example, the test case(s) that lead to the discovery of the fault. This information may be used to determine which types of test cases are more efficient than others.
- *If the fault has been corrected* – yes or no.

The type of fault and the fault description may be difficult to fill out when a failure is first discovered. This information may be added while correcting the fault causing the failure. If the tester of a unit is not able to reproduce a failure, a fault should still be reported [21].

The impact and the type of a fault are two classifications of faults. The classifications should strive to be [44]:

- Orthogonal – every fault should belong to exactly one class.
- Clear – different testers should classify a fault into the same class.

The impact of a fault is the impact or consequence of the fault [26]. The fault impact classes for the UTM will be (based on Perry [42] and ahamad [3]):

1. Critical – The fault results in failures that stop the software system, a software subsystem or a software unit from operating. For example, a fault that leads to an infinite loop so that the system does not respond is a critical fault.
2. Major – The fault results in failures that cause an incorrect, incomplete or inconsistent output. For example, a fault in an algorithm that causes the output result to be ten times too large is a major fault.
3. Minor – The fault does not result in failures that cause an incorrect output. For example, a fault in the unit's documentation.

The impact of a fault may change when more information is discovered about a fault's consequence.

Various classifications of types of failures exist. According to Kelly and Shepard [27] many organizations use only the impact classes to classify faults. However, to achieve a better understanding of the effectiveness of software development methodologies, a more detailed classification of faults is required [27]. The fault type classification used for the UTM is based on three classifications, IBM's Orthogonal defect classification (ODC) [22], Hewlett-

Packard's fault classification (as described in Pfleeger [44]) and a variant of the ODC called ODC-CC (Orthogonal defect classification for computational code) [27], with a starting-point in the Hewlett-Packard approach. The classes valid for unit testing have been identified from the three classifications, and the identified classes have been combined to produce the following classes:

1. *Calculation/logic* – Faults concerning for example the correctness of algorithms, predicate expressions or error handling.
2. *Data handling* – Faults concerning for example initialization and assignment of data or the use of correct data types.
3. *Interface* – Faults concerning for example arguments of class operations or functionality of class operations.
4. *Support code* – Faults concerning for example unit test drivers or stubs.
5. *Design/specification* – Faults concerning for example return types of class operations or wrong number of states of a class. Although similar to the faults in the 'Interface' class, observe that in the 'Design/specification' class the faults are in the design or in the specification of the unit.
6. *Other* – Faults that either seem to belong to more than one of the other classes, or do not fit into any of the other classes.

Faults that seem to belong to the 'Design/specification' class should be discussed with the unit designer/specifier before reporting, to make sure that the design/specification has not been misunderstood.

Because the fault type classification for the UTM is a synthesis of three other fault type classifications, it may be questioned whether the UTM fault type classification is orthogonal and clear. By analyzing the faults in the 'Other' class, an evaluation of how orthogonal the UTM fault type classification is may be done. To evaluate if the UTM is clear, classification of the same faults must be performed by different testers.

3.4.28.2 Unit test reports

A test report "summarizes all outcomes of testing" [43]. The unit test report for a unit should include (based on SEMM [62], Perry [42] and Pol et al. [45]):

- *Unit description* – Description (e.g. name) of the unit the test report concerns. Should include version number if applicable.
- *Have the test goals for the unit been reached* – Yes or No.
- *Last date unit test cases were run* – When the unit was last tested.
- *Test level goals for the unit* – How much testing was planned to be done for the unit, e.g. what level of coverage was to be achieved?
- *Test level reached for the unit* – What level of testing was achieved for the unit, e.g. what level of coverage was achieved?
- *Total number of test cases for the unit*
- *List of faults (ID) that have not been corrected yet* – Should include the reasons why the faults have not been corrected.
- *References to associated test plan(s) and test cases* – References to the test plan and the test cases concerning the unit.
- *System used* – The computer system and operating system used when the test cases were last performed.

The number of test cases may be used to derive metrics. A list of faults that have not been corrected yet may be important if the person working with the unit is replaced or if different persons are working on the unit.

How unit test reports should be documented will be included in the UTM.

3.5 Summary

The following methods and other unit testing process related activities discussed in this chapter were included in the UTM document [63] (The UTM document is an internal Westinghouse document and is not available outside of Westinghouse. Also, the content of the UTM document in the future may not correspond to the content at the moment of writing this thesis):

- A unit in object-oriented unit testing is primarily a class, but may also be a cluster of interdependent classes.
- Unit tests should primarily be designed during the test planning activity and executed during the implementation activity.

- One person should have all the roles of unit test case designer, unit test case executer and unit implementer.
- Black-box unit testing methods: equivalence partitioning, boundary value analysis and the all-pairs method.
- White-box unit testing methods: path testing, multiple condition testing, loop testing and data flow testing.
- Methods for testing exceptions.
- Unit test methods specific to object-oriented code: state-based unit testing and methods described concerning inheritance, polymorphism and information hiding. Also, methods for selecting an advantageous order of unit testing in a class are included.
- Unit test drivers for classes should be separate classes, and should be organized as an image of the inheritance hierarchy of the system tested. Test cases should be implemented as operations in test classes and the test object should be passed as an argument to the test case operations.
- Stubs should be avoided if possible. If stubs are used, the stubs should be unit tested, since they may evolve to the real implementation.
- Risk analysis should be used to prioritize which classes should be unit tested and how much the classes should be unit tested. Important parts of the system that are commonly used are more important than parts which are only rarely exercised, and black-box unit testing methods should be prioritized over white-box unit testing methods.
- The goals for how much a unit should be tested should be based on coverage strategies and criteria based on faults and duration of testing.
- It is better to have too many test cases than to few, but redundant test cases should if possible be avoided.
- Test cases should be documented and test results should be reported, which includes fault reports.

4 Case study

4.1 Introduction

This chapter explains how the UTM developed in Chapter 3 was applied, in a case study, on a selected unit that is a part of the production control system in the rod manufacturing workshop at the Westinghouse fuel factory in Västerås. The chapter explains how the case study was set up and how the case study was executed. The goals of the case study were to find useful software tools that could be integrated with the UTM, and to evaluate the methods in the methodology. A secondary goal was to cover as many paths through the code as possible to find faults. The case study may be described as in Figure 4.1. The case study contains a setup and an execution phase. In the setup phase a software tool research was performed and a suitable unit to be tested during the case study was selected from the production control system used in the fuel factory at Westinghouse. In the execution phase of the case study the tools found during the tool research together with the UTM were applied on the selected unit. The results from the case study are presented in Chapter 5, and will be compared to the expectations established in Section 4.2.

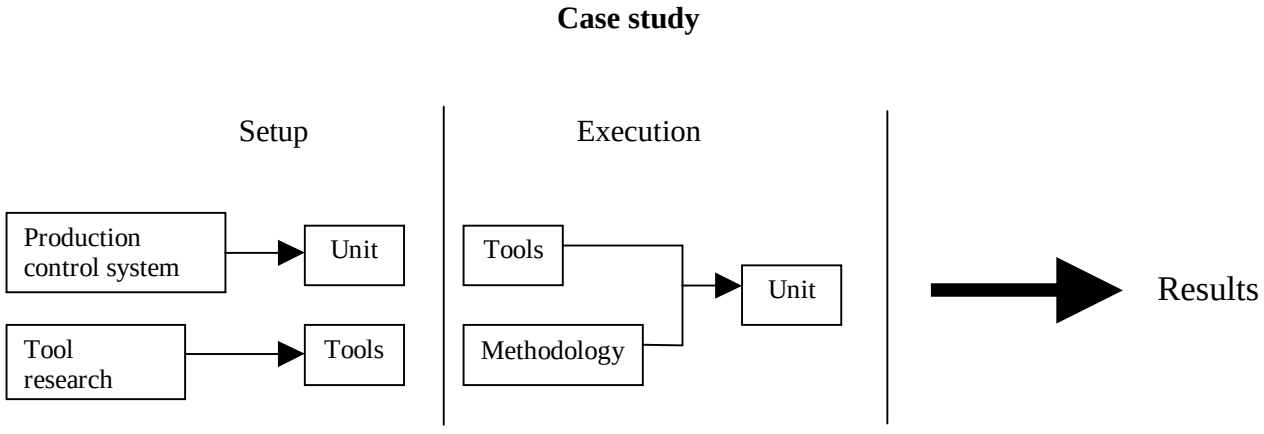


Figure 4.1: Overview of the case study

4.2 Case study expectations

The purpose of the case study was to evaluate the UTM. The following points were the expected results from the case study:

- All of the unit testing methods used are performable, in the sense that it is possible to design and execute test cases by using the instructions in the UTM document [63].
- Faults are found.
- White-box testing category methods discover faults in code that black-box testing methods for the same code do not reveal.
- Black-box testing category methods discover faults in code that white-box testing methods for the same code do not reveal.
- Each unit testing method reveals a fault.
- Every test case should be documented according to the UTM.
- Every fault should be documented according to the UTM.
- The test results for the Rodscanner unit should be documented according to the UTM.
- Private members should be tested using the “friend” keyword.
- The needs for extension and clarification of the UTM should be established.
- At least branch coverage should be reached for all code in the Rodscanner class.

4.3 Case study setup

The purpose of the case study setup phase was to gather and prepare all resources needed to execute the case study. The case study was performed using Visual Basic .NET since the code tested was written in the same language. Therefore, Visual Studio 2003 was installed on the computers to be used in the case study. Templates to be used for test documentation were constructed in Microsoft Excel. The software tools and the unit used in the case study are described in the following sections.

4.3.1 Tool research

Before the case study was executed, a software tool search was performed in order to find suitable tools for the case study.

Recall from Section 3.3 that one of the requirements of ManIT is that all unit tests should be automated, preferably using a software tool. There exist a number of software tools that automate tests. The tools included in the “test automation tools” search were evaluated against the following criteria:

(The tool should:)

- act as a unit testing framework or be specific for unit testing.
- be easy to use.
- include a graphical user interface.
- be possible to integrate in Visual Studio .NET.
- be documented.

A number of different software tools have been ported from JUnit [34], which is a testing framework for Java. JUnit may be used in Java software development to automate unit tests. MbUnit [20], csUnit [57], dotUnit [49], NUnit [39] and ZaneBug [2] are all variants of JUnit that are similar to each other and may be integrated in a .NET environment. NUnit and dotUnit were selected and compared in the case study because NUnit is a software tool already in use at ManIT and dotUnit was the tool that differed most from NUnit. For the results of the comparison, see Section 5.5.

When using the basis path testing method described in Section 3.4.9 flow graphs are preferably used to identify independent paths. No software tool was found that automatically draws flow graphs. However Visustin [5] is a software tool that automatically draws flow charts. Flow charts are closely related to flow graphs and may manually be translated to flow graphs with less effort than translating code to flow graphs. A demo version of Visustin has been used in the case study together with basis path testing to automatically draw flow charts, which then have been translated to flow graphs. The flow graphs were drawn using Microsoft Word and Microsoft Paint. Example of a flow chart drawn by Visustin is provided in Figure 4.2.

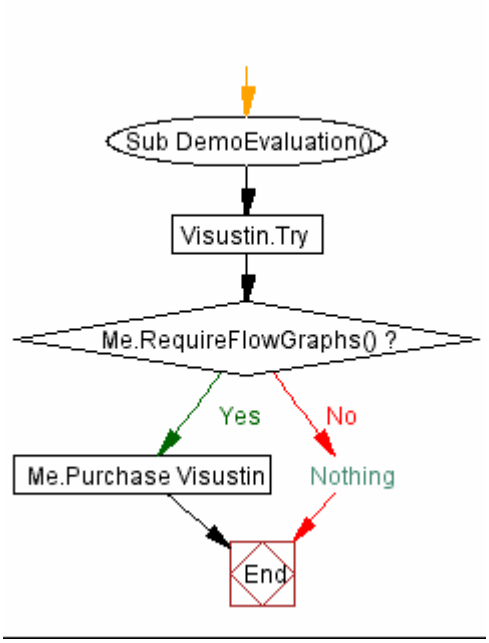


Figure 4.2: Visustin flow chart example

Project Analyzer [4] is a complete code review and quality control tool for Visual Basic. In the case study, Project Analyzer was used to automatically calculate the cyclomatic complexity value for different operations.

The code coverage software tools found were tools NCover [36], NCoverView [31] and CoverageEye.NET [19] that measured statement coverage, and Clover.NET [14] that measured branch coverage. NCover was a command-line tool that could be executed from a GUI with the help of NCoverView. However, NCoverView was not available for download during the case study. CoverageEye.NET was downloaded but the case study performers were not able to make the program work. Clover.NET worked and was the only tool that measured branch coverage, and was therefore chosen to be used in the case study.

4.3.2 Test unit

The test unit used in the case study was a Visual Basic .NET class named Rodscanner. Rodscanner handles communication between a production control system in the rod manufacturing workshop and the machine that scans fuel rods.

The Rodscanner class was not part of any inheritance hierarchy, did not involve any polymorphism and did not maintain any states, implying that the only object-oriented methods that could be applied were the information hiding method and the order of testing method. The fact that the inheritance, polymorphism and state-based testing methods could not be evaluated is however in accordance with the statement that the whole UTM should not be adopted at once, discussed in Section 3.2. Instead, these object-oriented methods will be

tested when more of the production control system is tested (see Section 6.3 about future work).

The class Rodscanner has ten different operations. The hierarchy of these operations may be viewed in Figure 4.3 where the arrows indicate operations used by other operations. The operation “Öppna” uses “LpMottag”, which uses ApMeddelande etc. “SerialIo” is a class that Rodscanner depends on. SerialIo was implemented as a stub during the case study execution. Five more stubs, Rs232, LoggFasad, DsLogg, DsProcess and Gen, were created but were mainly present to make the code compile.

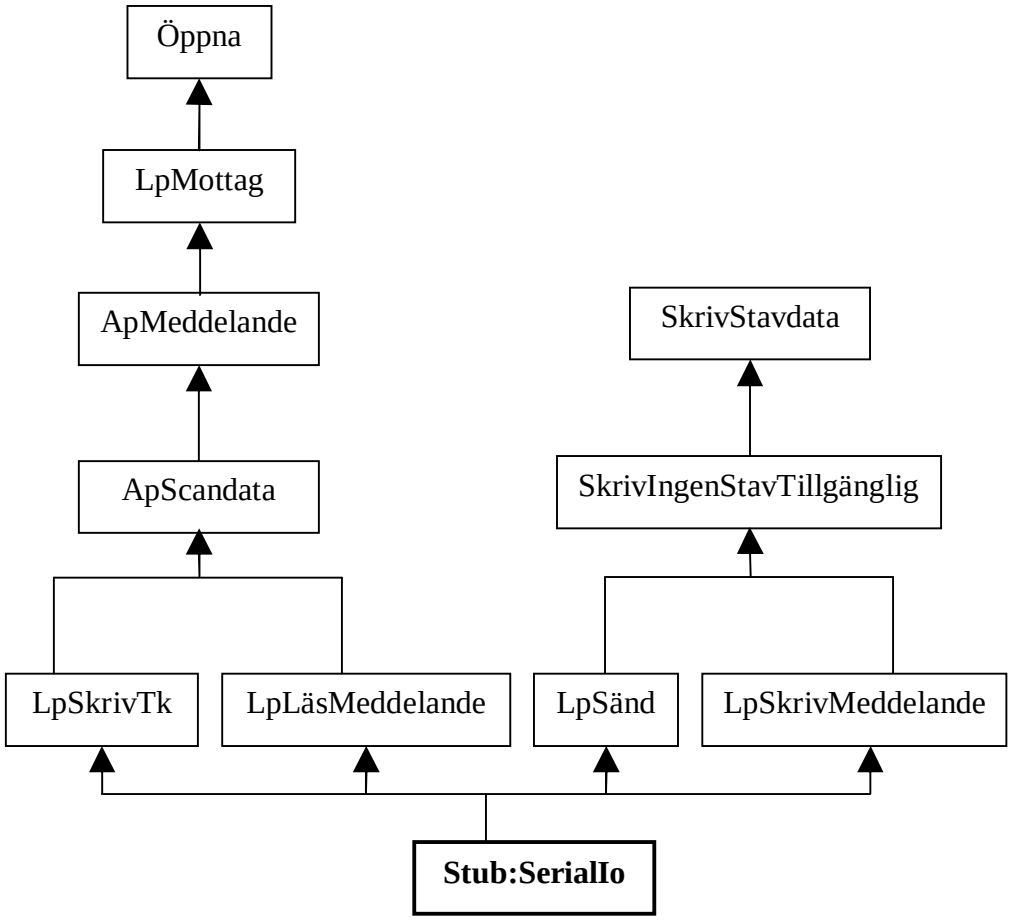


Figure 4.3: Operation hierarchy overview of class Rodscanner.

The exchange of information between the rodscanner machine and the production control system may be described as in Table 4.1.

Rodscanner machine	Message	Production control system
Need rod information		
	-- Rod data request -->	
		Retrieve rod data from DB
	<-- Rod data --	
Rod scanning		
	-- Scan data -->	
		Scan data stored in DB

Table 4.1: Information exchange between the rodscanner machine and the production control system.

4.4 Case study execution

The testing was performed in a bottom-up order, meaning that the private operations that did not depend on any other operations in the Rodscanner class were tested first. See Figure 4.3 for the dependence hierarchy of the Rodscanner operations.

The design of the test cases was documented using an excel document template. Additional sheets in the excel document were created to store graphs, tables and additional information for each of the operations tested. The designed test cases for an operation were categorized by the unit testing method used. The unit testing methods for testing polymorphism and inheritance, and the unit testing method state-based testing were not performed during the case study because the methods were not applicable on the class Rodscanner. Also, it was not possible to evaluate the practice of designing and implementing test cases before the unit code is written, since the case study was performed on an already existing software system.

For each operation tested, the steps visualized in Figure 4.4 were followed.

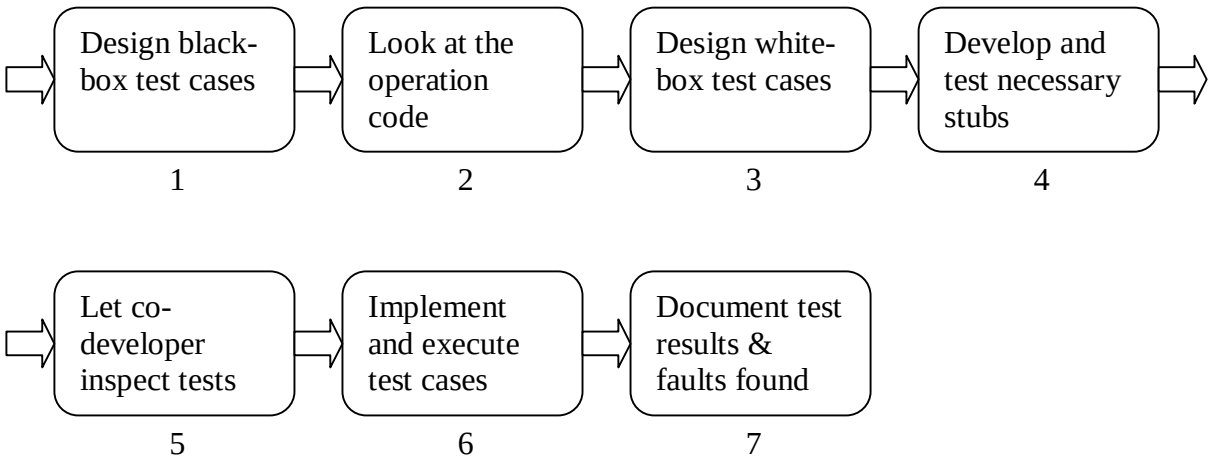


Figure 4.4: Case study execution steps

When an operation was to be tested, the first methods used were the black-box testing category methods. The black-box test cases were designed by looking at the specification text of an operation and also by looking at the Rodscanner interface document [58]. Equivalence partitioning, boundary value analysis and the all-pairs method were used when applicable. Then the code was inspected and carefully analyzed as the white-box category test cases were designed. The basis path method, described in Section 3.4.9, was used for every operation, and loop testing, multiple condition testing and data flow testing were used when applicable. The guidance of when to use data flow testing (mentioned in Appendix I.1, cyclomatic complexity < 10, maximum of 6 variables) was followed.

Inspections of test cases were performed, but no formal inspections according to SEMM [62] were performed because resources for a formal inspection were not available. The inspections showed that all test cases were documented according to Section 3.4.22. For examples of test case documentation, see Appendix G.

The designed test cases were implemented as test drivers using Visual Basic .NET code in the class RodscannerTest. Visual Studio 2003 was used as the programming IDE. The Rodscanner class was located in a project called “Handlers”, so a new project was created called “HandlersTest”. The Rodscanner class was linked in, while the test classes and the stubs were created in the new project (see Figure 4.5).

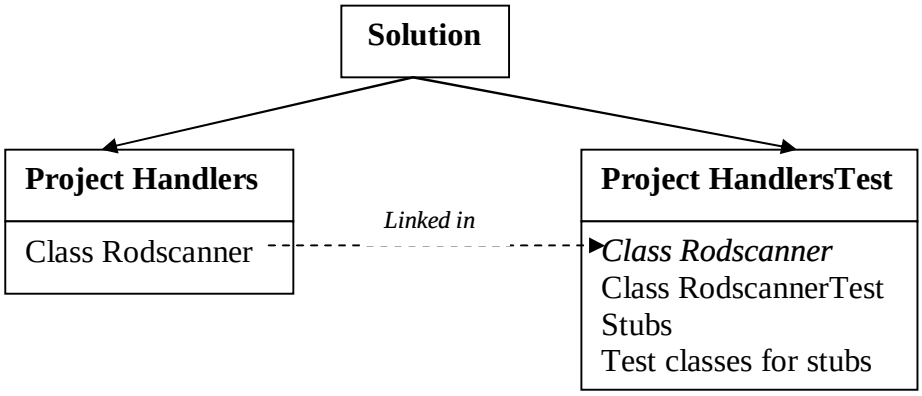


Figure 4.5: Visual Studio solution organization

The test case drivers were executed using dotUnit and NUnit. One of the authors of this thesis used dotUnit and the other author used NUnit and the results from using the tools are included in Section 5.5. The person that designed a particular set of test cases also implemented and executed those particular test cases.

If two or more different methods in the UTM generated the same test cases, the redundant test cases were not removed. Instead, a full set[1] of test cases were designed and implemented for each method used, so that the ability of each method to reveal faults could be determined. Redundant test cases generated within the same method were not documented and implemented because these redundant test cases did not provide any additional information about the relative ability of the methods to reveal faults. An example of redundant test cases within the same method is provided in the multiple condition table in Appendix G.

The criteria for ending unit testing included that all methods that were applicable for an operation were used, and used in accordance with the recommendations in the UTM document [63].

Test cases were implemented that exercised all blocks of code that caught exceptions. There was no specification of which exceptions might be thrown, so a general exception was used.

The faults found were described, categorized and reported in the Excel document.

When trying to use the keyword “friend” to test private operations, it was discovered that “friend” has a different meaning in Visual Basic .NET than what was expected and assumed in the UTM (described as a problem in Section 6.1.2). Instead, the .NET library System.Reflection was used. A class named “PrivateHelper” was developed (and tested in

[1] A full set of test cases in the context mentioned above is the maximum number of test cases generated by a specific method in the unit testing methodology.

accordance with the UTM) that wraps the details from the System.Reflection library and provides static operations that may be used to invoke non-public operations and to get and set non-public fields of other classes. The implications of not being able to use the “friend” keyword were that time was spent on developing the “PrivateHelper” class, and that testing non-public operations required a somewhat more complex syntax than would have been possible if the “friend” keyword had worked as expected.

Six stubs were created to test the class Rodscanner. The largest and most used stub, SerialIO, was tested using only black-box testing with equivalence partitioning and boundary value analysis. The reason for not testing the SerialIO stub more was that testing the Rodscanner class was prioritized. The stubs Rs232, LoggFasad, DsLogg, DsProcess and Gen were not tested, as they were mainly present to make the code compile.

5 Results and evaluations

5.1 Introduction

Section 5.2 in this chapter discusses the case study results compared to the case study expectations. Time estimate calculations for future work based on the case study results are provided in Section 5.3. The following section, 5.4, presents the results of the case study and an evaluation of the UTM. Finally, Section 5.5 includes a comparison between the two unit testing framework tools used in the case study, NUnit and dotUnit.

The operation “LpSänd” is used as an example throughout this chapter since many of the unit testing methods were applied on LpSänd. The code for LpSänd is provided in Appendix F.

5.2 Results against case study expectations

The expectations from Section 4.2 are repeated below together with the results for each expectation.

- *All of the unit testing methods used are performable, in the sense that it is possible to design and execute test cases by using the instructions in the UTM.*

  As may be seen in Table 5.6 in Section 5.4.9.1, test cases were designed and executed for all unit testing methods but inheritance, polymorphism and state-based testing, which were not applicable in the case study.

- *Faults are found.*

  28 faults were found. Table 5.1 shows the faults with the fault type classes defined in Section 3.4.28.1. Notice that every fault type class except the fourth and the sixth class contains faults. Faults belonging to fault type class four, “Support code”, were however found when testing the SerialIO stub. The nine test cases designed and the two faults found when testing the SerialIO stub were not included in the statistics in

this section since all methods applicable were not used when testing the SerialIO stub (see Section 4.4).

Fault type class	Faults
1 – Calculation/logic	18
2 – Data handling	3
3 – Interface	6
4 – Support code	0
5 – Design/specification	1
6 – Other	0
Total # of faults	28

Table 5.1: Faults found during case study

- *White-box testing category methods discover faults in code that black-box testing methods for the same code do not reveal.*
- *Black-box testing category methods discover faults in code that white-box testing methods for the same code do not reveal.*

  Black-box testing revealed faults that white-box testing did not and vice versa.

- *Each unit testing method reveals a fault.*

  Each unit testing method used in the case study revealed a fault except multiple condition testing, which may be seen in Table 5.6 in Section 5.4.9.1. Although multiple condition testing did not reveal any faults, the method is useful when the "all predicate uses" coverage described in Section 3.4.26 should be reached.

- *Every test case should be documented according to the UTM.*

  Test cases were documented in a Microsoft Excel sheet in compliance with Section 3.4.22. See Section 5.4.3 for more details.

- *Every fault should be documented according to the UTM.*

  Faults were documented in a Microsoft Excel sheet in compliance with Section 3.4.28.1. See Section 5.4.9.1 for more details.

- *The test results for the Rodscanner unit should be documented according to the UTM.*

  The test results for the Rodscanner were documented in a Microsoft Excel sheet in compliance with Section 3.4.28.2. See Section 5.4.9.2 for more details.

- *Private members should be tested using the "friend" keyword.*

  The keyword "friend" in Visual Basic .NET did not have the intended functionality and is described as a problem in Section 6.1.2. Instead, a class PrivateHelper was developed that uses the System.Reflection library.

- *The needs for extension and clarification of the UTM should be established.*

  All results and evaluations described in this chapter will together lead to an extended and clarified UTM. Future work on the UTM will include the extensions and clarifications.

- *At least branch coverage should be reached for all code in the Rodscanner class.*

According to the code coverage tool used in the case study, Clover.NET [14], branch coverage was reached for all code in the Rodscanner class. Higher coverage was reached for individual operations, as presented in Section 5.4.7.

5.3 Time estimates

The time required to perform the case study and unit test the class Rodscanner may be used in calculating time estimates for future unit testing. Note that the time estimates in this section is based on the time required to perform the case study, in which the UTM was used for the first time. It is likely that less time is required for unit testing when the UTM has been used for a longer period of time. The time required for testing the class Rodscanner may have been shortened by prioritizing which parts that should be tested, but no prioritizing was done. Also, the unit testing was performed on an already existing system, i.e. the code that was tested needed to be comprehended before unit testing began. If the unit testing was to be performed during development and the same person had the three roles described in Section 3.4.3, the code would already be known and no time would have to be spent on understanding the code.

During the case study, testing the class Rodscanner required 80 man hours. Recall from Section 5.4.8, that 21 % of the test cases produced were redundant. Then it may be expected

that (79 % of 80 ≈) 63 man hours are required to test the class Rodscanner. The class Rodscanner included 459 lines of code[1] that implies (459 / 63 ≈) 7.3 lines of code were tested per hour. The project Handlers in which the class Rodscanner is included contains 8617 lines of code (excluding the code for the class Rodscanner). The time required to test the project Handlers would be (8617 / 7.3 ≈) 1200 man hours.

The time may be shortened by such factors as experience in using the UTM and better tools (for instance, better tools for automatic construction of flow graphs). If enough time is still not available, prioritization among the units will help decide what and how much to test.

5.4 Methodology usage results and evaluation

In this section, the results of using the UTM are presented and evaluated. The sections below correspond to the sections in Chapter 3 involving the methods and other unit testing process related activities included in the UTM document. Test case code examples implemented using the unit testing methods are presented in Appendix H.

5.4.1 Traditional unit testing methods

5.4.1.1 Equivalence partitioning and boundary value analysis

When an operation was to be tested during the case study, equivalence partitions and boundary values were identified. If the specification for an operation was incomplete or vague, the partitions and boundary values were difficult to identify. Test cases were designed and implemented to test the partitions and boundaries according to Section 3.4.6.

Equivalence partitioning and boundary value analysis were mostly used for black-box testing but was also useful for white-box testing. As an example, when testing an independent path throughout an operation the values controlling the path could be chosen on boundaries.

Equivalence partitioning and boundary value analysis are two methods that are essential to reduce the number of test cases needed for an operation. Consider the operation LpSkrivTkn in the class Rodscanner, to which a Byte parameter is passed. To check all values that could be passed to the operation would require 256 test cases (all valid values for a variable of type Byte). Instead, only test cases were designed to test the inputs 0, 1, 254, 255 and a typical value in between the boundaries.

[1] Every line in the source code files was counted as a line of code.

5.4.1.2 All-pairs testing

The all-pairs method was used to reduce the number of test cases at three occasions. Twice together with equivalence partitioning and boundary value analysis, and one time together with multiple condition testing. The percentages of reduction were:

- 12 out of 24 possible test cases designed: 50% reduction of number of test cases.
- 40 out of 640 possible test cases designed: ~94% reduction of number of test cases.
- 9 out of 27 possible test cases designed: ~67% reduction of number of test cases.

Table 5.2 shows one of the all-pairs matrices used in the case study. The matrix corresponds to the first point in the bullet list above with a 50% reduction of number of test cases. Each row in Table 5.2 represents a test case. All combinations of values of the

Test case ID	Antal kasskoder	Stav OK	Mode	Tested
036	0	Y	A	X
037	0	N	M	X
038	1	Y	M	X
039	1	N	A	X
040	2	Y	A	X
041	2	N	M	X
042	3	Y	M	X
043	3	N	A	X
044	4	Y	A	X
045	4	N	M	X
046	5	Y	M	X
047	5	N	A	X

Y = Yes, N = No, A = Automatic, M = Manual

Table 5.2: Example of an all-pair matrix used in the case study.

variables “Antal kasskoder”, “Stav OK” and “Mode” (that is, pairwise combinations) are represented in at least one test case (See Section 3.4.8 for details on how to create an all-pairs matrix).

Since the all-pairs method was used successfully together with multiple condition testing, the conclusion was that the all-pairs method may be used not only in a black-box context together with equivalence partitioning and boundary value analysis, but also in white-box contexts, together with multiple condition testing.

5.4.1.3 Basis path testing

Compared to the other unit testing methods, basis path testing (together with data flow testing) was the most difficult method learning to use. There exist difficulties with identifying independent paths though a flow graph (see Section 3.4.9). Experience in using the basis path testing method decreased the time required to use the method.

The first step in the basis path testing method was to automatically draw a flow chart using the software tool Visustin [5] (described in Section 4.3.1). The flow chart was then translated into a flow graph by drawing the graph in Microsoft Paint or Microsoft Word. Drawing flow graphs in Word and Paint was time consuming. More specialized applications for drawing flow graphs are required to decrease the time needed for using the basis path testing method. Automatic calculation of the cyclomatic complexity of the operations was performed using the software tool Project Analyzer [4]. As a final step the number of independent paths identified, using the flow graph, was checked against the value of the cyclomatic complexity.

When performing the basis path testing method after other methods had been performed during the case study, a number of redundant test cases were produced. Considering the time and effort required to use the basis path testing method, and the fact that identifying the redundant test cases produced by the basis path testing method was more difficult than identifying redundant test cases when using other testing methods, it may be questioned if basis path testing should be performed after other methods have been performed. Further usage of the UTM included in future work (included in Section 6.3) may evaluate whether instead the basis path testing method should be performed before other methods.

5.4.1.4 Multiple condition testing

Multiple condition testing was experienced as one of the easier methods in the UTM to perform. When knowing how to create a truth table, the multiple condition test cases could be derived directly from the truth table.

When the number of test cases in the truth table would require more time to implement than what was available, the all-pairs method was used to reduce the number of test cases.

Multiple condition testing without using the all-pairs method assures that the multiple condition coverage level has been reached (see Section 3.4.26 for details on the coverage levels).

5.4.1.5 Loop testing

There where two main observations made from performing loop testing during the case study. Firstly, when loop testing was performed on loops that always executed a determined number of times, such as a for loop iterating through an array with a predefined size, the test cases generated were always covered by other unit testing methods. The conclusion is that when there is no way to control how many times a loop is executed, there is no need to perform loop testing since other test cases cover the loop.

Secondly, the translation of designed test cases to implemented test cases for loop testing was not always as straightforward as for e.g. multiple condition testing. As an example, the loop test cases for the operation LpSänd required the SerialIO stub to be extended to provide additional control of the behavior of the stub. The additional functionality of the stub was only used by the loop test cases.

5.4.1.6 Data flow testing

As mentioned in Section 5.4.1.3, basis path testing and data flow testing, compared to the other methods used in the case study, were the most difficult methods to learn to use. Data flow testing requires a flow graph drawn by hand on paper (see Appendix I for guidelines). Visustin [5] (described in Section 4.3.1) was used to automatically draw a flow chart, which then was translated to a flow graph on paper. The guidelines by Binder [11] included in Appendix I were then followed. According to the guidelines, test cases should be made that include as many definition-use paths as possible. In some cases during the case study, the paths constructed according to the guidelines included 63 nodes (including loops) in the flow graph. As the number of nodes included in a path increase, the complexity of the test case including the path grows and the probability that the test case will contain faults increases. Therefore, a path that includes a large number[1] of nodes should be divided into several paths.

The data flow testing method was only used when the cyclomatic complexity value was less than 10, and the operation included at most 6 variables, as described in Appendix I.1. If individual positions in arrays were used or defined in the code, each position was counted and treated as an individual variable.

In most cases during the case study, data flow testing and basis path testing generated equivalent test cases. Further usage of the UTM included in future work (see Section 6.3) may

[1] How large the number of nodes should be will have to be determined by further usage of the methodology, included in future work (Section 6.3).

evaluate which of the two methods is appropriate to use when testing a specific operation, or if only one of the two methods is sufficient to use.

5.4.1.7 Testing exceptions

The class Rodscanner that was tested during the case study did not throw any explicit exceptions and therefore no test cases were designed to handle exceptions thrown by Rodscanner. The class Rodscanner handled two exceptions thrown by a class that Rodscanner used. Therefore, test cases were designed to check that these exceptions were handled correctly by the class Rodscanner. No "catchall" construct (described in Section 3.4.14) was needed because unexpected exceptions were caught and displayed by the unit testing frameworks used in the case study, NUnit and dotUnit.

5.4.2 Unit testing methods for object-oriented code

The Rodscanner class was not part of any inheritance hierarchy, did not involve any polymorphism and did not maintain any states. Therefore, inheritance, polymorphism and state-based testing have not been evaluated.

5.4.2.1 Information hiding

The access of non-public operations and members of the class Rodscanner was enabled by implementing a class named PrivateHelper (based on operations originally created by Tim Stall [51], described in Section 6.1.2), which used the Microsoft library System.Reflection. The keyword "friend" in Visual Basic .NET may not be used to give only the class RodscannerTest access to the class Rodscanners' private operations and members. Usage of the keyword "friend" is described as a problem, described in Section 6.1.2.

5.4.2.2 Order of testing within a unit

Section 3.4.21 describes the preferred order of testing within a unit. In the Rodscanner class there was only a default constructor and no getters or setters. Only single operations were tested, and there was no inheritance involved. Therefore, the only recommendation from Section 3.4.21 that was applied was to test helper operations before operations using the helper operations and testing in a bottom-up order. The other recommendations in Section 3.4.21 will be evaluated when testing more of the production control system (see Section 6.3 about future work).

The operations of the Rodscanner class were tested in a bottom-up order in accordance with Figure 4.3. The fact that operations lower in the hierarchy were already tested when an

operation was tested should increase the probability that failing test cases were due to faults located in the operation being tested. In fact, all the faults found during the case study were located in the operation tested at the moment, not in any helper operations lower in the hierarchy used by the operations being tested.

5.4.3 Documentation of unit test cases

During the case study, test cases were documented together in a Microsoft Excel sheet template, which was developed in compliance with Section 3.4.22. A Microsoft Excel sheet was selected as appropriate documentation of test cases. The decision was based on Patton's [41] and Hutcheson's [21] recommendations described in Section 3.4.22, and the fact that no database was available to store the information.

An excerpt from the test case documentation is included in

Table 5.3 on the next page. The complete test documentation for the operation "LpSänd" is provided in Appendix G.

At the beginning of the case study, information categories in the test case documentation template were difficult to separate. But as the case study proceeded, the test case documentation template became more comprehensible, and the information categories were easier to understand. All information categories in the template were relevant, especially the "additional information" category, which was used frequently to clarify test cases. Division of the test case information into categories simplified inspection of the test cases.

Additional sheets were created in the same Excel file as the test case documentation to store flow graphs, flow charts, independent path tables, truth tables and all-pairs matrices (examples of these additional sheets are provided in Appendix G). Each test case associated with an additional sheet was provided with a reference to the sheet.

Test case ID	Test Item	Description	Input		Result state	Additional information
			Setup	Actions		
	LpSänd					
	Equivalence partitioning/Boundary value analysis	Identified using equivalence partitioning and boundary value analysis. These tests and several others are also covered by the black-box tests for LpSkrivMeddelande				
022	LpSänd	Test with message size 1 and boundary value content hex00	message = [00], set serialIOStub to return OK and ACK	LpSänd(message)	0 returned	
023	LpSänd	Test with message size 20 and boundary value content hex01, hex7F, hex00	message = [20,7F,00,20,7F,00,...], set serialIOStub to return OK and ACK	LpSänd(message)	0 returned	

Table 5.3: Excerpt of test case documentation

5.4.4 Unit test drivers

Test drivers were implemented in separate test classes as described in Section 3.4.23. Since the class Rodscanner was not a part of an inheritance hierarchy, the parallel test class hierarchy strategy, also described in Section 3.4.23, was not applied in the case study.

When implementing test cases based on the documented design of the test cases “the code wrote itself”, i.e. the experience was that the translation of the test case design to implementation was easy. As an example of the translation from design to implementation, the design of the test case with ID 033 is shown in Table 5.4, and the implementation of the same test case is shown in Figure 5.1. The implementation is performed using Visual Basic .NET and NUnit [39] functionality (for a description of NUnit, see Section 5.5). To understand the design and implementation of test case ID 033 the reader is encouraged to check the code provided in Appendix F for the operation that is tested; LpSänd.

The benefits of separating the test case design from the implementation found were:

- Attention is given to the test case, not the implementation.
- Test cases that resemble each other may be identified before the implementation of the test cases. Helper operations that contain common code for the test cases that resemble each other may then be implemented and used in test classes that include the test cases.

Test case ID	Test Item	Description	Input		Result state	Additional information
			Setup	Actions		
033	LpSänd, while loop	6 iterations	set serialIOstub to first return ReturnCode.Timeout two times, then ReturnCode.OK and ACK	LpSänd(message)	ReturnCode.OK returned	content of message does not matter

Table 5.4: Test case design for test case ID 033.

```
<Test(), Category(LPSÄND)> _
   Public Sub testCaseID033()

      'innehållet i meddelande spelar ingen roll
      m_meddelande = New Byte() {&H33}

      m_serialIOstub.m_returnSeveralCodes = True
      m_serialIOstub.m_codesToReturn = New SerialIo.ReturnCode()
                     {SerialIo.ReturnCode.Timeout, _
                                 SerialIo.ReturnCode.Timeout, _
                                 SerialIo.ReturnCode.OK}

      m_serialIOstub.m_readWaitBuffer = New Byte()
      {PrivateHelper.getStaticField(m_rodscanner.GetType, ACK)}

      Assert.AreEqual(CType(WE.PRS.HANDLER.Rodscanner.ReturKod.OK, Integer), _
      PrivateHelper.runInstanceMethod(LPSÄND, m_rodscanner, m_meddelande))

   End Sub
```

Figure 5.1: Test case implementation for test case ID 033

Test case execution was performed using the two unit testing frameworks dotUnit [49] and NUnit [39]. A test class, including test cases, was included in the respective unit testing framework. To exercise the test class' test cases a button in the respective graphical user interface was clicked and all test cases were executed. If a failure occurred, the GUI displayed information about the failure (e.g. a stack trace or an exception description) and where the failure was located (which class, operation and line of code). A comparison between dotUnit and NUnit is included in Section 5.5.

5.4.5 Stubs

Stubs were used in the case study according to the recommendations described in Section 3.4.24. The stub SerialIo, which was the most frequently used stub in the case study, included generation of exceptions and checking of messages passed from the class Rodscanner. To control and check the behavior of the class Rodscanner in the stub SerialIo, the interface of the stub SerialIo had to be altered from the interface of the real implementation. Additional public members were added to the stub. These additional public members were then used in the test class, RodscannerTest, to easier control and observe the behavior of the class Rodscanner.

In compliance with the UTM, test cases were designed to test the only stub that included implementation, SerialIo.

The stub SerialIo included 223 lines of code[1] while the real implementation of SerialIo included 231 lines of code. It may seem logical to draw the conclusion that the real implementation of SerialIo should have been used instead of the stub, considering the effort needed to create the stub. However, in the case study, the stub was necessary to both isolate the tests for the class Rodscanner and to check and control the behavior of Rodscanner. The real implementation of the class SerialIo was not possible to use to control the behavior of Rodscanner.

Using stubs in the case study caused a major problem to appear. The problem is described in Section 6.1.3.

5.4.6 Prioritizing unit tests

Since the Rodscanner class was recommended for testing by ManIT, prioritizing among classes was not done in the case study. As more of the production control system will be

[1] Every line in the source code files was counted as a line of code.

tested (see Section 6.3 about future work), prioritizing among classes will be necessary and an evaluation will be possible.

Black-box testing was prioritized before white-box testing in the sense that black-box testing was performed before white-box testing.

5.4.7 Criteria for ending unit testing

Since the time for the case study was limited, and since each unit testing method in the UTM was to be used fully (i.e. generating the maximum number of test cases) whenever applicable, the main criteria for ending the unit testing was based on time rather than coverage. The Rodscanner class was to be tested as much as possible during the time of the case study. A secondary goal was that at least branch coverage should be reached for all the operations that were tested.

The time for the case study was sufficient to test all the operations in the Rodscanner class using all the methods applicable for each operation. Branch coverage was reached for the 10 operations of the Rodscanner class. A higher level of coverage than branch coverage was reached for all operations in the class Rodscanner. Referring to Figure 3.16,

- all-paths was reached for 1 operation,
- all definition-use paths was reached for 5 operations and
- all uses was reached for 4 operations.

To determine that branch coverage was reached for all operations, a software tool called Clover.NET [14] was used. Clover.NET is an add-in tool for Visual Studio. When using Clover.NET, the code is compiled to a special folder and extra code is added that monitors how the original code is exercised. The code compiled by Clover.NET is then executed. The results may then be loaded in Visual Studio, showing the percentage of statements and branches that has been covered for each operation in a class. Code that has not been exercised is marked red by Clover.NET. Figure 5.2 shows an example of coverage measurement with Clover.NET. Notice that Figure 5.2 illustrates an example only, and does not represent the results of the case study.

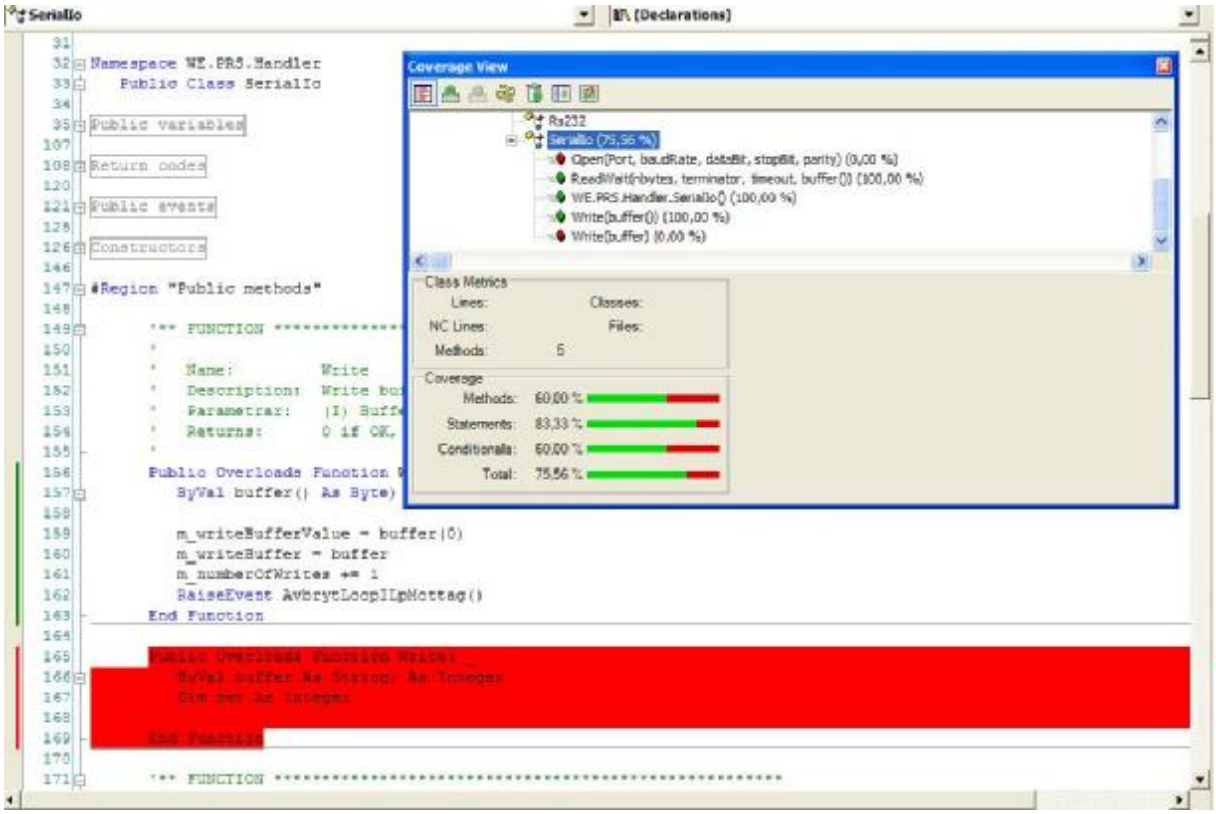


Figure 5.2: Example of Clover.NET coverage measure results

The higher levels of coverage reached for the operations of the Rodscanner class were determined by hand, by examining the paths exercised by the test cases. The reason for determining levels of coverage by hand was that no tool was found that was able to measure levels of coverage higher than branch coverage. A more efficient way of determining coverage levels higher than branch coverage will have to be developed.

5.4.8 Redundant test cases

As mentioned in Section 4.4, redundant test cases were not avoided in the case study so that the ability of each method to reveal faults could be determined. Of all 182 test cases implemented in the case study, 38 of the test cases were redundant, i.e. 21 percent of the test cases implemented were redundant. In other words, the case study produced a number of redundant test cases that may be avoided in future testing, and the time required to test a unit may be reduced.

5.4.9 Reporting unit test results

5.4.9.1 Fault reports

In Table G.3 in Appendix G the fault report for the LpSänd operation is shown. The experiences from the case study concerning the fault report in general were:

- It was at first difficult to know exactly what should be written under the columns “Situation of occurrence”, “Effect of the fault” and “Fault description”. It did however become clearer what to write in the columns as the experience of designing test cases grew. The descriptions of the three columns provided in the fault report template should be improved to further clarify what to write in the respective columns.
- A new column named “Location” should be added to the fault report. Instead of, as done in the case study, writing the location of the fault in the “Situation of occurrence” column, the new column should be used. The “Situation of occurrence” column should be used to specify the actions taken that revealed the fault, i.e. a description of how to reproduce the effects of the fault. The new column makes it easier to search for faults located in a specific operation, and also gives the information in the “Situation of occurrence” column a higher degree of cohesion.
- The description of the column “Situation of occurrence” should explain that it is not mandatory to fill out the column. For instance, when the fault is “declared but unused data objects”, the “Situation of occurrence” information is not applicable.

For the convenience of the reader, the fault impact and fault type classes defined in Section 3.4.28.1 are here repeated. The fault impact classes in the UTM are:

1. *Critical* - The fault results in failures that stop the software system, a software subsystem or a software unit from operating.
2. *Major* – The fault results in failures that cause an incorrect, incomplete or inconsistent output.
3. *Minor* – The fault does not result in failures that cause an incorrect output.

The fault type classes in the UTM are:

1. *Calculation/logic* – Faults concerning for example the correctness of algorithms, predicate expressions or error handling.
2. *Data handling* – Faults concerning for example initialization and assignment of data or the use of correct data types.
3. *Interface* – Faults concerning for example arguments of class operations or functionality of class operations.
4. *Support code* – Faults concerning for example unit test drivers or stubs.
5. *Design/specification* – Faults concerning for example return types of class operations or wrong number of states of a class.
6. *Other* – Faults that either seems to belong to more than one of the other classes, or does not fit into any of the other classes.

The experiences from the case study concerning the fault impact classes showed that the fault impact classes used were not specifically designed for reporting faults found during unit testing. During the case study, it was possible to classify the faults found into the fault impact classes, but to do so, an estimation of the effect the fault might have on the system had to be done. Further use of the UTM and fault impact classification is needed to be able to adapt the fault impact classes to better suit faults found during unit testing.

To evaluate the experiences from the case study concerning the fault type classes, first consider that the amount of documented design and specification for the operations in the Rodscanner class varied between the operations. The lack of documentation made it difficult to determine if a fault should be classified as fault type 3, “Interface”, or fault type 5, “Design/Specification”. In other words, was the operation implemented incorrectly or was the operation designed/specified incorrectly? The conclusion is that the fault type classification is easier to use when the design and specification of the unit tested is clearly documented. The results concerning the orthogonality and clarity of the fault impact and fault type classifications were:

- No fault was classified in the fault type class 6, “Other”. Since no fault was classified in the “other” fault type class, the fault type classification may be orthogonal. However, since only one class in one specific system was tested, the faults found may be specific for the Rodscanner class. More faults from a wider spectrum of classes and systems need to be classified with the fault type

classification in the UTM to determine if the fault type classes are indeed orthogonal.

Concerning the orthogonality of the fault impact classes, the experience of the case study performers was that the classes need to be better or perhaps even differently specified, since it was difficult to determine whether a fault would lead to incorrect output or not. How the improvement of the fault impact classification is to be done will have to be evaluated in future work, when more faults have been classified.

- To evaluate the clarity of the two fault classifications, the two case study performers classified the faults found by each other and then compared how the faults were classified. Of the 28 faults found in the case study, only 2 were classified into different fault type classes by the two case study performers. This indicates that the fault type classification is clear. For the fault impact classification, 8 of the 28 faults found were classified differently. This indicates that the fault impact classes need to be further clarified. However, more faults need to be classified, by different individuals, to better evaluate if the fault classifications in the UTM are clear, and if not, how to improve the classifications.

The faults found in the case study are shown in Table 5.5, where the impact classes defined in Section 3.4.28.1 are also included.

Fault impact class	**Faults**
1 - Critical	0
2 - Major	15
3 - Minor	13
Total # of faults	28

Table 5.5: Faults in each impact class

A total of 28 faults were found. 12 of the 15 faults classified as fault impact class 2 were due to missing input parameter value checking, i.e. allowing values that were not specified for the operation.

Table 5.6 shows information about the faults found classified by unit testing method.

Testing method (and inspections)	# of test cases	# of faults found
Equivalence partitioning / Boundary value analysis	96	8
All-pairs testing	61	6
Basis path testing	33	2
Loop testing	17	2
Multiple condition testing	24	0
Data flow testing	7	1
Additional tests	5	0
Inspection	N/A	20
Total	182[1]	28[2]

Table 5.6: Faults found by each unit testing method

The equivalence partitioning method and the boundary value analysis method have been grouped together since they were always used together in the case study. The row "Additional tests" represents test cases that were written but could not be classified into any of the unit testing methods of the UTM. All five of these additional tests may be described as performing equivalence partitioning and boundary value analysis on internal input values of an operation, that is, using equivalence partitioning and boundary value analysis with a white-box approach. The row "Inspection" lists the faults that were found not by the failure of any test case, but by inspecting the code when designing white-box test cases. Finding 20 faults by inspection indicates that the technique is important to use. As discussed in Section 3.4.3, the developer and inspector should not be the same person, because in that case the probability of implementing faults that are not discovered increases. Instead, the role of the developer and the inspector should be separated. The inspection technique belongs to the area of software inspection in contrast to the area of software testing, which this thesis is focused on. Therefore, the inspection technique is not included in this thesis but is an interesting subject to investigate and possibly include in the UTM in the future.

As a summary, by designing and executing the 182 test cases implemented, 28 unique faults were found.

[1] Since the all-pairs method test cases were always used together with other methods, see Section 5.4.1.2, the all-pairs test cases have not been included in the total sum.

[2] Faults that have been found by more than one method have only been counted once. Therefore the total number of faults is less than the sum.

Table 5.7 shows the faults classified by impact class and type class for the different unit testing methods and for inspection. The table is intended to reveal if any of the unit testing methods tends to discover more of a particular fault impact or fault type class.

Method	Fault impact class			Fault type classes					
	1	*2*	*3*	*1*	*2*	*3*	*4*	*5*	*6*
Equivalence / Boundary		5	3	6		2			
All-pairs testing		5	1	5		1			
Basis path testing			2	1		1			
Loop testing			2	1		1			
Multiple condition testing									
Data flow testing			1			1			
Additional Tests									
Inspection		10	10	12	3	4		1	

Table 5.7: Number of faults in each fault impact and fault type class found by each unit testing method

Three of the 20 faults found by inspection belonged to fault type class 2, “data handling”, and were found specifically when designing data flow test cases, after other white-box test cases had already been designed. The three faults were declared data objects that were never used. The other faults in the “Inspection” row were found when trying to understand the code before any white-box test cases were designed. This suggest that thinking in terms of how the data is used (“data-flow thinking”) when inspecting code may reveal data handling faults that are not discovered by test cases.

The fault in fault type class 5, “Design/Specification”, was found by inspection. However, the faults in fault type class 3, “Interface” were similar to the class 5 fault. Of all these similar faults, it was only one fault that was in the specification instead of in the implementation of the interface, and this one fault happened to be found by inspection. No conclusion that inspections are better at finding fault type class 5 faults should therefore be drawn.

5.4.9.2 Unit test reports

The unit test report for the Rodscanner unit is shown in Table G.4 in Appendix G. The experience of the case study performers were that all the fields in the unit test report were significant for reporting the results of the unit testing, except for the field ”Have the test goals for the unit been reached”. The information in this field could easily be determined by comparing the two fields “Test level goals for the unit” and “Test level reached for the unit”. The conclusion is therefore to remove the ”Have the test goals for the unit been reached” from the unit test report template.

5.5 DotUnit versus NUnit

As mentioned in Section 4.3.1, both dotUnit and NUnit are unit testing frameworks which may be used to automate tests. The following features are common to both tools:

- The installation includes documentation with images and code examples.
- Tests may be written in any programming language supported by .NET.
- Setup and TearDown are optional operations that are called, respectively, before and after each test case is run. In dotUnit the names of the operations should be specified exactly as in the previous sentence. In NUnit the operations may use the [SetUp] or [TearDown] attributes.
- Results from a test run may be saved to an xml file.
- Unexpected exceptions during a test run are reported to the user.
- Integration into Visual Studio .NET is possible. With dotUnit the integration is done by adding a reference to the file “dotunit.framework.dll” that is provided with the installation. With NUnit, the additional program TestDriven.NET [53] may be used to accomplish the integration.
- Test classes may be included in a test suite that runs all test cases contained in the test classes.
- Exceptions may be tested. In dotUnit by using a try-catch block and in NUnit by using the [ExpectedException] attribute.

DotUnit is closely similar to JUnit. DotUnit includes the following additional features:

- Each test class inherits the TestCase class of the dotUnit library and may contain a set of test cases.
- Each test case is an operation that begins with the word “Test”.
- There are two operations provided in dotUnit that assert the success or failure of a test case; AssertTrue(Boolean expression) or the operation AssertEquals(…). AssertEquals is overloaded for each simple data type. An optional string parameter to the assert operations may identify and categorize the tests.
- DotUnit includes functionality to access non-public operations within classes.
- The GUI runner may be run from within the .NET environment. A snapshot of the dotUnit GUI is provided in Figure 5.3.
- Specific test cases or groups of test cases may only be run by specifying the specific test cases in the code for the test runner. Normally, all test cases within a test class or tests suite are run at the same time. If only a few of these test cases should be run, then the test runner needs to be recoded.

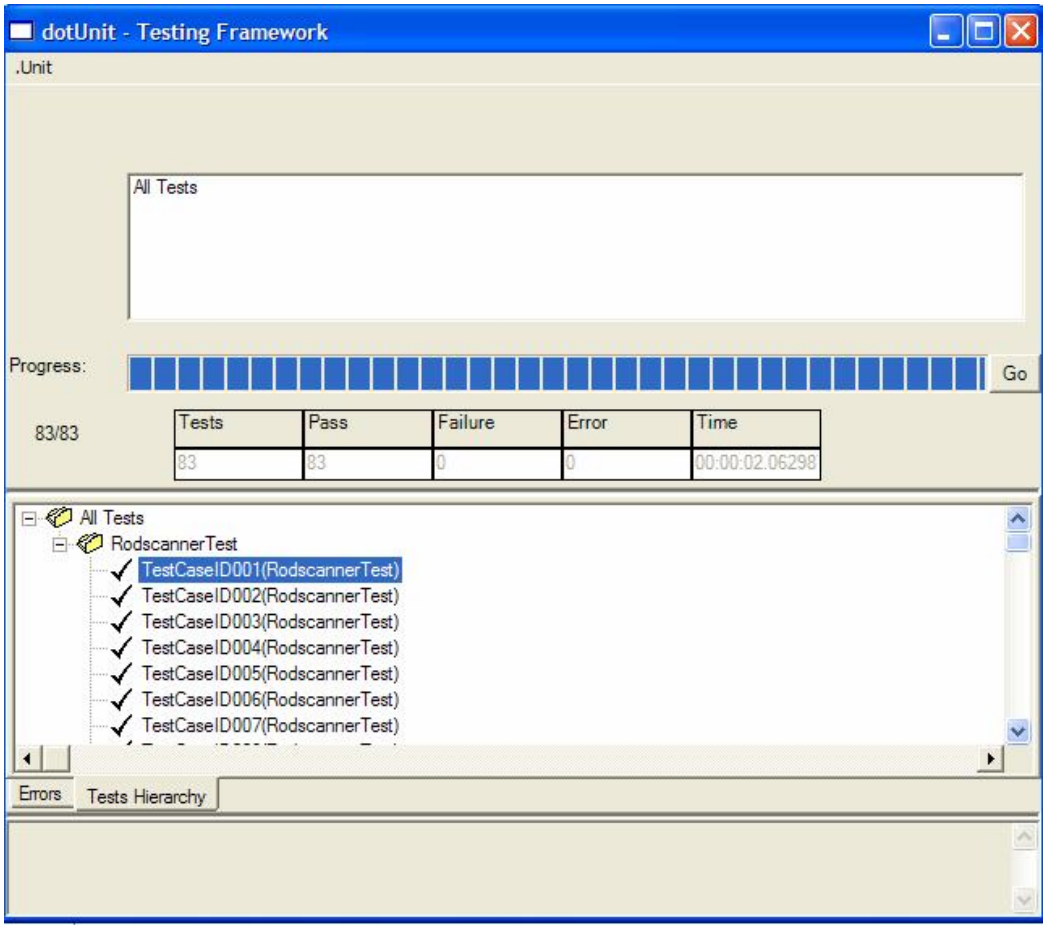


Figure 5.3: DotUnit Graphical User Interface

A simple example of a dotUnit test class coded in Visual Basic .NET is provided in Figure 5.4.

```
Public Class MyTest
    Inherits dotUnit.Framework.TestCase

    Private privateVariable As Integer

    Public Sub New(ByVal name As String)
        MyBase.New(name)
    End Sub

    Protected Overrides Sub SetUp()
        privateVariable = 1
    End Sub

    Public Sub TestCase1()
        AssertEquals(1, privateVariable)
    End Sub

End Class
```

Figure 5.4: DotUnit example in Visual Basic .NET code

NUnit includes the following additional features:

- A class containing test cases is marked with the attribute [TestFixture].
- Each test case is an operation marked with the attribute [Test].
- NUnit provides 8 operations that assert the success or failure of a test case, e.g. IsTrue, IsFalse, AreEqual and AreSame.
- With the [Category] attribute the test cases may be classified into categories that may easily be run separately from the NUnit GUI. A snapshot of the NUnit GUI is provided in Figure 5.5.
- Single test cases may be run from the NUnit GUI.

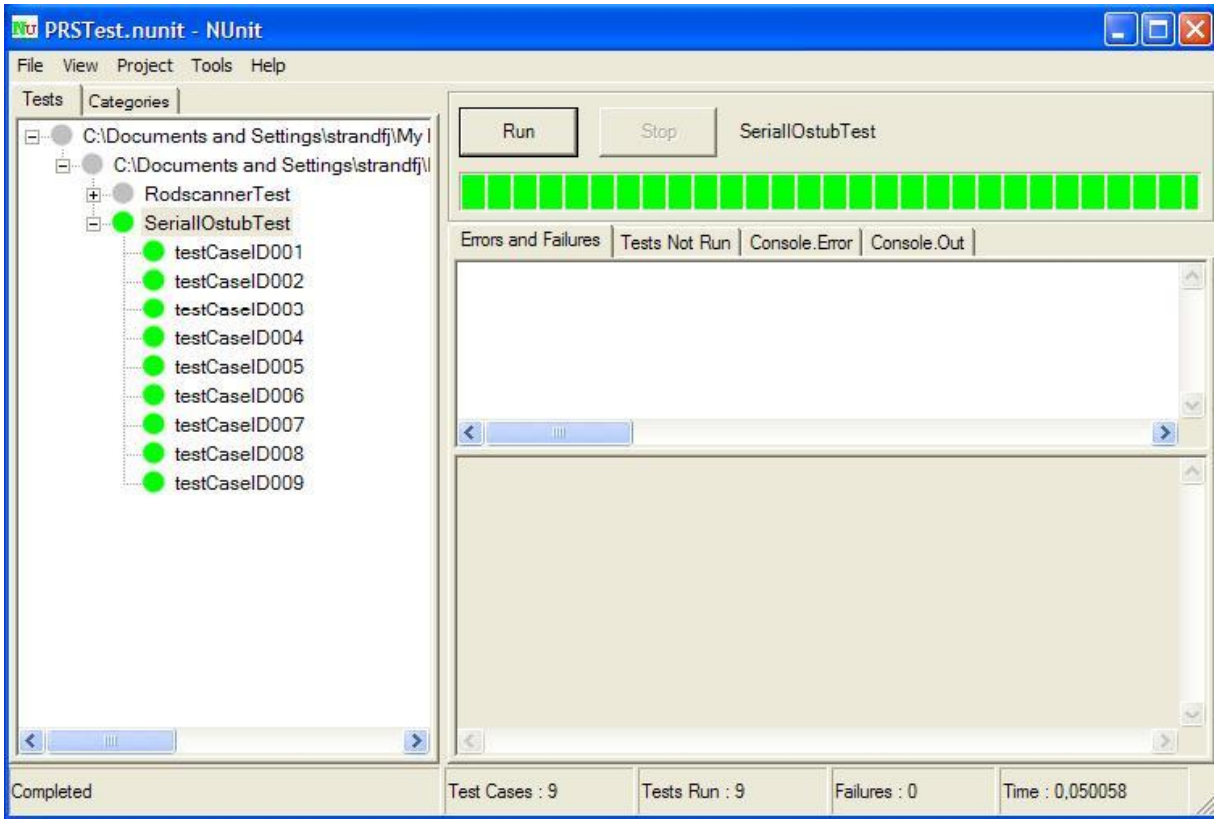


Figure 5.5: NUnit Graphical User Interface

The NUnit representation of the dotUnit test class from Figure 5.4 is provided in Figure 5.6.

```
<TestFixture()> _
Public Class MyTest

   Private privateVariable As Integer

   <SetUp()> _
   Protected Sub setup()
      privateVariable = 1
   End Sub

   <Test(), Category("myCategory")> _
   Public Sub testCase1()
      Assert.AreEqual(1, privateVariable)
   End Sub

  End Class
```

Figure 5.6: NUnit example in Visual Basic .NET code

As a summary, the features considered the most valuable during the case study are listed in Table 5.8.

Feature	**dotUnit**	**NUnit**
Possible to integrate with Visual Studio	yes	yes
Categorizing tests	yes	yes
Only run selected tests from GUI	no	yes
Setup operation	yes	yes
Teardown operation	yes	yes
Supports access to private operations	yes	no
Save test results as XML	yes	yes
Automatic handling of unexpected exception	yes	yes
Documentation available	yes	yes

Table 5.8: Summary of unit testing tools features

DotUnit supports accessing private operations, which NUnit does not. However, the PrivateHelper class developed during the case study (see Section 6.1.2) may be used to access not only private operations but all non-public (e.g. protected) members of a class. Therefore, the lack of support for private access in NUnit is not a problem.

In dotUnit, it is not possible to select and run single tests from the GUI. Since the NUnit GUI does allow single tests to be run, and the PrivateHelper class may be used for non-public access, NUnit was chosen to be the unit testing framework used in the UTM.

5.6 Summary

In this chapter, all methods and other unit testing process related activities developed in Chapter 3 have been evaluated (except the methods inheritance, polymorphism and state-based testing) using the results of the case study (described in Chapter 4). The evaluations and results indicate that the UTM is applicable to unit testing. However, in order to improve the UTM and bring the methods to a more practical level the methodology should be revised and extended according to the results. By using the UTM, higher coverage of the test unit, the class Rodscanner, was achieved than expected. Inspection and 182 test cases designed and implemented revealed 28 faults.

DotUnit and NUnit are two unit testing frameworks that were used during the case study to automate tests. By comparison and evaluation of dotUnit and NUnit, NUnit was selected as the superior unit testing framework and will be included in the UTM.

The results from the case study corresponded well to the expectations. However, two expectations were not in agreement with the results. Firstly, the use of the keyword “friend” in Visual Basic .NET did not have the intended functionality and is described as a problem in Section 6.1.2. Secondly, the unit testing method multiple condition did not reveal any faults. However, the multiple condition testing method is useful when the “all predicate uses” coverage should be reached.

Approximate time estimation for unit testing the project “Handlers” that contains the class Rodscanner comes to a total of 1200 man hours.

6 Conclusions

6.1 Problems

6.1.1 Learning a new programming language

In the case study Visual Basic .NET was the programming language used to implement test cases because the class that was tested was coded in the same language. The test case implementers (the authors of this thesis) were not familiar with Visual Basic .NET. Learning the new syntax and semantics of Visual Basic .NET was not a major problem but still caused a few errors that delayed an already tightly-scheduled case study plan. The most time-consuming problems were:

In Visual Basic .NET an integer array of 5 elements is declared in the following way:

Dim array(4) as Integer

To access the individual positions in the array indices from 0 to 4 are used. In the C-based languages that the authors of this thesis were used to, the array would be declared as:

int[] x = new int[5];

The indices would be the same in the C example. The problem was that to declare an array of size 5 in Visual Basic .NET the number 4 had to be used.

In the documentation of NUnit all examples were in C# code. Time consuming efforts were required searching the Internet to find out how to write NUnit attributes in Visual Basic .NET. In C#, NUnit attributes for test case operations are written as follows:

```
[Test]
[Category("myCategory")]
public void testCase001()
```

In Visual Basic .NET the same attributes are written:

<Test(), Category("myCategory")> _
Public Sub testCase001()

6.1.2 Information hiding

During the case study, a problem associated with information hiding in the class that was tested appeared. Recall from Section 3.4.20 that to access non-public members and operations of a class the keyword "friend" should be used. If friend is not available in a specific programming language, the .NET System.Reflection library should be used. In Visual Basic .NET, the keyword "friend" is available. However, an entity in Visual Basic .NET with "friend" access is accessible within the entire program that contains the entity declaration [1]. Consider a class named MyClass and a test class for MyClass named MyClassTest. The intent of using the keyword "friend" was to give MyClassTest access to MyClass' non-public members and operations. In the programming language C++ MyTestClass may be declared as friend in MyClass. The declaration gives MyTestClass access to MyClass' non-public members and operations. In Visual Basic .NET, MyClass would have to be declared "friend" and this declaration would give all other classes contained within the same program access to MyClass' non-public members and operations, which is not desirable.

In the case study, the System.Reflection library was used to gain access to non-public members and operations of the class that was tested. The System.Reflection library includes a number of operations that are not involved with accessing non-public members and operations, and the number of parameters passed to operations in the System.Reflection library may be reduced by concentrating the functionality on non-public access. A wrapper class called PrivateHelper was implemented that simplified the use of the System.Reflection library and concentrated on non-public access functionality. The class PrivateHelper is based on operations originally created by Tim Stall [51]. The use of PrivateHelper solved the problem with information hiding of the class that was tested during the case study.

6.1.3 Stubs

A major problem that occurred during the case study was the inclusion of stubs in Visual Studio .NET projects. In the case study, a Visual Studio .NET solution named "UnitTests" was created (see Figure 6.1).

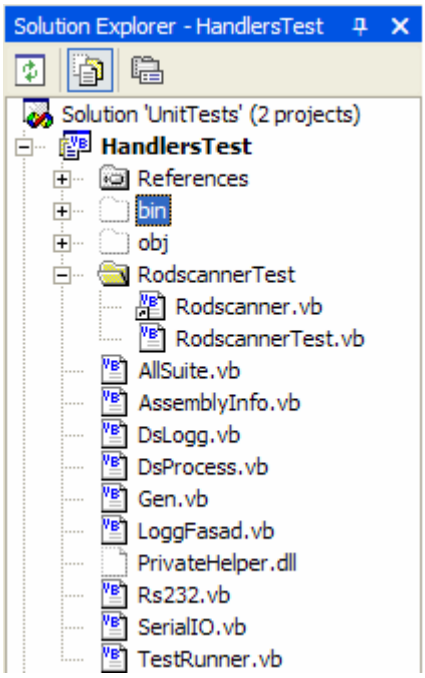


Figure 6.1: The Visual Studio .NET solution used in the case study.

The class that was tested in the case study was the Rodscanner class. As may be seen in Figure 6.1, the Rodscanner class file is linked into the project HandlersTest from the "real" project named Handlers in a solution called PRS (thereby the project name HandlersTest, which is intended to contain all test classes for units that are located in the Handlers project). The Rodscanner class is linked into the HandlersTest project rather than copied so that changes in the Rodscanner class do not need to be implemented in two files. The class RodscannerTest contains all test cases for the class Rodscanner. Classes DsLogg, DsProcess, Gen, LoggFasad, Rs232 and SerialIO, located in the HandlersTest project are all stubs. The class AllSuite (used with dotUnit) is intended to collect all test classes and the TestRunner class runs the dotUnit GUI.

The problem occurs when the "real implementation" of a stub is included in the project HandlersTest. As an example, since the "real" SerialIO is located in the project Handlers, like the Rodscanner class, a test class for SerialIO should be included in the HandlersTest project and the real SerialIO should be linked in, like the Rodscanner class. The stub for SerialIO may not be removed because the stub is used in the automated tests for the Rodscanner class to control and check the behavior of the Rodscanner class. If the real implementation of SerialIO were to be included in the project HandlersTest, a conflict would occur between the SerialIO stub and the real SerialIO because they are located in the same namespace. In other words, if the real implementation of a stub is included in the same project as the stub, a namespace conflict will occur.

A solution to the problem is to isolate the conflicting classes into different Visual Studio .NET solutions, but this is not practical and the problem needs to be solved in a better way.

6.2 Conclusions

6.2.1 Conclusions based on the case study results

The conclusions based on the case study results are that the UTM was usable for unit testing in the case study, but that further usage and evaluation is needed to evaluate all methods and to improve the UTM.

Adjustments to the UTM according to the results of Chapter 5 are necessary. An example of such an adjustment is to add a column “Location” in the unit fault report template, as described in Section 5.4.9.1. Also, the problem with stubs (see Section 6.1.3) needs to be solved or at least somehow avoided.

6.2.2 ManIT requirements

In Section 3.3 the requirements from ManIT on the UTM were described. Next, conclusions of how well the UTM fulfilled the requirements are provided:

- Compatibility with the .NET development environment

  It was possible to use the UTM in the .NET environment. The UTM is only specific to .NET in the sense that it is recommended to use the System.Reflection library to overcome information hiding, when the “friend” keyword is not available in the programming language used. The rest of the UTM is not specific to any development environment.

  The only problem encountered when applying the UTM together with .NET was the problem of organizing stubs in Visual Studio .NET, described in Section 6.1.3.

- Unit testing object-oriented software systems

  The UTM contains both traditional unit testing methods, that may be used when testing the operations of classes, and methods specific for testing object-oriented systems.

- Compatibility with Visual Basic .NET

The UTM was applicable for both testing code that was written in Visual Basic .NET, as well as for implementing the test cases in Visual Basic .NET. The problem with the semantics of the keyword “friend”, described in Section 6.1.2, was solved by using the class PrivateHelper developed during the case study.

- Generic methodology

  The UTM was developed to be applicable for testing object-oriented code in the .NET environment. The UTM is specific for testing object-oriented code, since unit testing methods for object-oriented code are included in the methodology. Such methods as implementing the test drivers as classes and organizing the test drivers as a parallel test class hierarchy would not be possible for non object-oriented code. However, all the traditional unit testing methods that are included in the UTM may be used for testing procedural code, since they may be applied on the operations of a class.

  The .NET environment did not set any restrictions on the UTM other than that the PrivateHelper class using the .NET System.Reflection library should be used. The class PrivateHelper should only be used when the programming language does not contain a language construct such as “friend” in C++ that allows a class to give unlimited access rights to a single other class. All other methods and other unit testing process related activities in the UTM document are generic concerning the development environment used.

  The UTM may be used with all programming languages that support object-orientation and that either contains a language construct with the same semantics as “friend” in C++, or that is supported by the .NET environment so that the PrivateHelper class may be used.

- Automated execution of unit tests

  Automatic execution was possible using both NUnit and dotUnit. NUnit was the unit testing framework selected to include in the UTM.

- Effect of the development group size

  The effect of having a small development group such as the one at ManIT (currently 5 developers) was that the UTM should not contain too many administrative procedures, such as writing and updating various documents. Any such procedures

should be aided by technology, i.e. software tools. Detailed instructions of work routines, i.e. methods in the UTM, should be included.

The UTM requires the test cases, test results and faults found to be documented, preferably using a database specialized for test documentation or if no database is available a spreadsheet. In the case study there was no database available so the documentation was done using a Microsoft Excel document, which contained individual sheets for the different types of documentation. The experience of the case study performers was that during the case study, using Excel sheets was practical. When more of the production control system at Westinghouse is tested using the UTM (see Future work in Section 6.3), it will be possible to evaluate if spreadsheets are useful also when there are a larger number of Excel documents stored in different locations, or if a database for test documentation will be necessary to make it easier to access all the information.

Concerning detailed instruction, the UTM did not include enough detail. More guidelines and examples of how to perform the methods in the UTM are needed. Detailed instructions will be developed in future work (see Section 6.3).

6.2.3 ManIT structured testing process

Existing documents such as SEMM are not detailed enough concerning unit testing to be practically useful. The UTM developed in this thesis is more detailed and contains more guidelines for unit testing than SEMM.

Section 2.6 describes how the testing process at ManIT is unstructured since no formal standards, procedures or guidelines are being used. The UTM document [63] may be seen as a standard containing procedures (methods) and guidelines. Therefore, if the UTM is used by ManIT, the methodology will contribute to a more structured testing process. The intention of the staff at ManIT is that the UTM will be used.

So far, the UTM has contributed to better quality of the production control system in that the class Rodscanner is now better tested. However, the faults found in the case study will have to be corrected to achieve the largest increase in quality from using the UTM.

The UTM developed in this thesis is a first step for ManIT towards a structured testing process. Although improvements and extensions of the UTM are necessary, ManIT is satisfied with the results of this thesis.

By using the methods and other unit testing process related activities in the UTM developed in this thesis, anyone (e.g. a company) with similar requirements for a UTM as

MIT, and that performs unit testing in an unstructured way, may benefit in that a more structured unit testing process is achieved.

6.3 Future work

Future work on the UTM developed in this thesis is already planned and is going to take place during the first six months of 2006. Currently, the UTM document [63] is a too theoretical document that does not include sufficient guidelines that in detail instruct how the methods should be performed. Therefore, guidelines together with explaining code examples will be developed and included in the UTM document in order to enhance the practicality of the document.

In the case study, only a single unit from the production control system in the rod manufacturing workshop at the Westinghouse fuel factory in Västerås was tested. Test methods for object-oriented code included in the UTM; inheritance, polymorphism and state-based testing, were not evaluated during the case study. A test plan will be developed that includes testing of additional units and evaluation of the test methods for object-oriented code. Testing of the additional units will be performed using the UTM. The testing is intended to find faults in the production control system and to further evaluate the UTM.

The methodology is concentrated on unit testing. A request from ManIT is that integration testing and GUI testing should be included in the methodology. The purpose is to enable the methodology to be used with types of testing other than unit testing.

References

[1] *4.6 Accessibility*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbls7/html/vblrfvbspec4_5.asp, December 7th, 2005.

[2] Adapdev Technologies, LLC. *Adaptev technologies, LLC – Zanebug*. http://www.adapdev.com/zanebug/, December 6th, 2005.

[3] ahamad. *Defect severity and defet priority*, October 24th, 2005. http://www.qasoftwaretesting.com/article36.html, November 3rd, 2005.

[4] Aivosto. *Project Analyzer v7.1 for Visual Basic, VB .NET and VBA.* http://www.aivosto.com/project/project.html, December 6th, 2005.

[5] Aivosto. *Visustin – Flow chart generator for VB, C#...*http://www.aivosto.com/visustin.html, December 6th, 2005.

[6] ASPAlliance.com. *Chapter 3: Testing.* http://authors.aspalliance.com/chapters/0672322331/0672322331_ch03.aspx, October 27th, 2005.

[7] Barker, Richard. *CASE*Method – Tasks and Deliverables.* Addison-Wesley, 1990.

[8] Beck, Kent. *Extreme Programming explained – embrace change*. Addison-Wesley, 2000.

[9] Beizer, Boris. *Software testing techniques.* New York: Van Nostrand, 2nd edition, 1990.

[10] Bell, Doug, Ian Morrey and John Pugh. *Software engineering – a programming approach.* Prentice Hall, 2nd edition, 1992.

[11] Binder, Robert V. *Testing object-oriented systems – models, patterns and tools.* Addison-Wesley, 2000.

[12] Bipin, Joshi. *Introduction to .NET Reflection.* http://www.dotnetbips.com/displayarticle.aspx?id=35, October 27th, 2005.

[13] Caspersen, Michael E., Madsen, Ole Lehrmann, Skov, Stefan Helleman. *Testing Object-Oriented Software – COT/2-43-V1.0*, February 28th, 2001. http://www.cit.dk/COT/reports/reports/Case2/43/cot-2-43.doc. October 19th, 2005.

[14] *Cenqua: Clover .NET Code Coverage.* http://www.cenqua.com/clover.net/, December 16th, 2005.

[15] CoderSource.net. *C# .Net Tutorial Reflection*, January 5th, 2004. http://www.codersource.net/csharp_tutorial_reflection.html, October 27th, 2005.

[16] Eklund, Sven and Hans Fernlund. *Programkonstruktion med kvalitet – projekthantering och ISO 9000.* Studentlitteratur, 1998.

[17] *Exception handling – Wikipedia, the free encyclopedia.* http://en.wikipedia.org/wiki/Exception_handling, October 25th, 2005.

[18] Georgetown University. *Data Warehouse – Glossary*. http://uis.georgetown.edu/departments/eets/dw/GLOSSARY0816.html, October 12th, 2005.

[19] GotDotNet. *GotDotNet User Sample: CoverageEye.NET*. http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=881a36c6-6f45-4485-a94e-060130687151, December 12th, 2005.

[20] *Home – confluence*. http://www.mertner.com/confluence/display/MbUnit/Home, December 6th, 2005.

[21] Hutcheson, Marnie L. *Software testing fundamentals.* Wiley, 2003.

[22] IBM Research. *Details of ODC v 5.11.* http://www.research.ibm.com/softeng/ODC/DETODC.HTM, November 3rd, 2005.

[23] *ICH Glossary*. http://www.ichnet.org/glossary.htm, October 12th, 2005.

[24] IEEE. *IEEE Standards Collection: Software Engineering.* IEEE Standard 610.12-1990, 1993.

[25] Jacobson, Ivar et al. *Object-oriented software engineering – a use case driven approach.* Addison-Wesley, 4th revised printing, 1992.

[26] Kaner, Cem, James Bach and Bret Pettichort. *Lessons learned in software testing.* Wiley, 2002.

[27] Kelly, Diane and Terry Shepard. *A case study in the use of defect classification in inspection.* http://portal.acm.org/citation.cfm?id=782103. Royal Military College of Canada, 2001.

[28] Kruchten, Philippe. *The rational unified process – en introduction.* Addison-Wesley, 2nd edition, svenska utgåvan, 2002.

[29] Lindegren, Håkan. *Programvaruprojekt – stabilitet, användbarhet och inkrementell utveckling.* Studentlitteratur, 2003.

[30] *Liskov substitution – Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Liskov_substitution_principle, October 26th, 2005.

[31] Mark Levision. *NCover and NCoverView – free code coverage tools for .NET*. http://dotnetjunkies.com/WebLog/mlevison/archive/2004/02/26/8024.aspx, December 12th, 2005.

[32] McCabe, T. "A software complexity measure", *IEEE Transactions on Software Engineering.* SE-2(4), pp. 308-20, 1976.

[33] McGregor, John D and David A. Sykes. *A practical guide to testing object-oriented software.* Addison-Wesley, 2001.

[34] Meade, Erik, Robert C. Martin and Jennifer Kohnke. *JUnit, Testing resources for Extreme programming*. http://www.junit.org/index.htm, December 5th, 2005.

[35] Miller, E., D.A. Steiner and G.J. Symons. *Testing tools in Encyclopedia of software engineering vol. II.* Wiley, 1994.

[36] *NCover – A test code coverage tool for C# .NET*. http://ncover.sourceforge.net. December 12th, 2005.

[37] Norbert, Oster. *Automated generation and evaluation of dataflow-based test data for object-oriented software*, SOQUA 2005, September 22nd, 2005. http://www.mathematik.uni-ulm.de/sai/jmayer/soqua05/slides/oster.pdf, December 22nd, 2005.

[38] Nordby, Eivind J., Blom Martin, Brunström, Anna. *A case study for an industry project.* Karlstad University, 1999.

[39] *NUnit – Home*. http://www.nunit.org. October 3rd, 2005.

[40] OPEN Process Framework Repository Organization. *Unit Testing.* http://www.donald-firesmith.com/index.html?Components/WorkUnits/Activities/Testing/UnitTesting.html~Contents, November 1st, 2005.

[41] Patton, Ron. *Software testing.* Sams publishing, 2001.

[42] Perry, William. *Effective Methods for Software testing*. John Wiley & Sons, 1995.

[43] Peters, James F and Witold Pedrycz. *Software engineering – an engineering approach.* John Wiley & Sons, 2000.

[44] Pfleeger, Shari Lawrence. *Software engineering theory and practice.* Prentice-Hall, International edition, 1998.

[45] Pol, Martin, Ruud Teunissen and Erik van Veenendaal. *Software testing – a guide to the TMap approach.* Addison-Wesley, 2002.

[46] Pressman, Roger S. *Software engineering – a practitioners approach.* MacGraw-Hill, 4th edition, 1997.

[47] Pressman. *Video 090 Understanding CASE (2) Software Quality Assurance.* http://notendur.unak.is/not/andy/Year%202%20Software%20Quality%20Assurance/Lectures/V090.pdf, October 19th, 2005.

[48] *Selecting Software Test Data Using Data Flow Information*. http://oregonstate.edu/~lawrancj/wiki/index.php/Selecting_Software_Test_Data_Using_Data_Flow_Information, December 22nd, 2005.

[49] Sepulveda, Christian. *dotUnit – Automated testing framework.* http://dotunit.sourceforge.net/, December 5th, 2005.

[50] Sommerville, Ian. *Software Engineering.* Addison-Wesley, 6th edition, 2001.

[51] Stall, Tim. *How to test private and protected methods in .NET,* March 1st, 2005. http://www.codeproject.com/csharp/TestNonPublicMembers.asp, December 7th, 2005.

[52] *TestComplete - Automated software testing tool*. http://www.automatedqa.com/products/testcomplete, October 3rd, 2005.

[53] *TestDriven.NET > Home*. http://www.testdriven.net, October 12th, 2005.

[54] *Testing Object Oriented Programs.* http://www.cs.mu.oz.au/342/Lectures/course-notes/chapter-6.pdf, October 12th, 2005.

[55] *The world wide web virtual library: the Z notation.* http://vl.zuser.org, October 17th, 2005.

[56] TimStall, The Code Project. *How to Test Private and Protected methods in .NET,* March 1st, 2005. http://www.codeproject.com/csharp/TestNonPublicMembers.asp, October 27th, 2005.

[57] *Welcome to csUnit.org!*. http://www.csunit.org/index.php, December 5th, 2005.

[58] Westinghouse. *Gränssnitt Rodscanner, BCT 04-081, revision 0.* Westinghouse internal document, February 11th, 2005.

[59] Westinghouse. *Nuclear fuel engineering procedure, EP-310 revision 15.* Westinghouse internal document, July 1st, 2005.

[60] Westinghouse. *Nukleär säkerhetsinformation.* Westinghouse external booklet, January 2004.

[61] Westinghouse. *Quality Management System revision 5.* Westinghouse internal document, October 1st, 2002.

[62] Westinghouse. *Software engineering methodology manual revision 32.* Westinghouse internal document.

[63] Westinghouse. *Unit testing methodology.* Westinghouse internal document.

[64] *Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Test_case, December 22nd, 2005.

[65] *WordNet Search – 2.1*. http://wordnet.princeton.edu/perl/webwn?o2=&o0=1&o6=&o1=1&o5=&o4=&o3=&s=methodology, October 12th, 2005.

[66] *WordNet Search – 2.1*. http://wordnet.princeton.edu/perl/webwn?o2=&o0=1&o6=&o1=1&o5=&o4=&o3=&s=method, October 12th, 2005.

A The concept of quality

A.1 Software quality

Patton [41] describes quality as “a degree of excellence”. A software product is of high quality if the software product meets the customer’s needs and the customer feels that the product is excellent and superior to his or her other choices. This definition suggests that software quality to a large degree is a subjective aspect, something that is in the eye of the beholder. One of the most important beholders is the customer.

Bell et al. [10] writes that software is of good quality if it

- meets its users needs.
- is reliable (does not fail).
- is easy to maintain.

So far, only the quality of the product has been discussed. Pfleeger writes about other aspects of quality, in particular, process quality. Process quality is [44]:

- To find faults as early as possible in the development process
- To determine as close as possible when particular faults are likely to occur
- To build in fault tolerance to minimize the likelihood that a fault will become a failure
- To find a number of alternative activities that may make the process more effective or efficient at assuring product quality

With this definition, a high level of process quality will help assuring a high product quality [29]. There are however other factors affecting the product quality too. According to Sommerville there are four factors, shown in Fig. A.1 [50]:

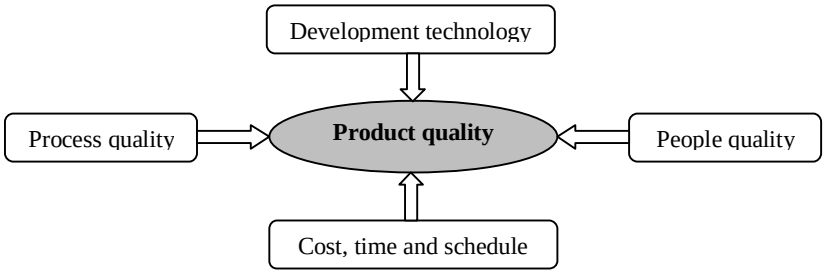


Figure A.1: Product quality factors [50]

The UTM developed in this thesis is intended to improve the process quality (specifically, *test* process quality) at ManIT. The development technology at ManIT will be improved, as appropriate tools will be used together with the UTM. “People quality” and “Cost, time and schedule” will not be treated in this thesis.

A.2 Quality assurance

The purpose of quality assurance is not only to find faults in software systems, but also to remove the faults. You could say that quality assurance is all the activities that are concerned with finding and removing faults, and also ensuring that the final product has been produced according to the requirements. “Quality assurance addresses activities designed to prevent defects as well as to remove those defects that do creep into the product.” [33]

B Non-software safety measures at Westinghouse Fuel Manufacturing

Since Westinghouse Fuel Manufacturing produces nuclear fuel, it is relevant to ask whether the software systems developed needs to be safety-critical or not. There is a high level of security and safety awareness and procedures at the industrial part of the fuel factory, which serves as an indicator of the importance of safety. The fuel factory does however not primarily depend on any software system for safety. Non-software safety measures include [60]:

- The uranium handled has a maximum enrichment of 5%. There has so far never been any nuclear accident at a factory handling this maximum enrichment.
- Prevention of interaction between the uranium in the containers is done by constructing the containers so that they may not be placed too close to each other.
- There are limits on how much uranium may be stored in the same area at the factory.

These points are some of the non-software safety measures at the factory, but there are several others [60].

There is also a Programmable Logic Controller (PLC) layer between the actual machines and the software systems developed and maintained by ManIT. The PLC systems contain safety controls. This means that a software failure is not a critical hazard at the factory.

C Software Engineering Methodology Manual (SEMM)

All information in this section, e.g. quotes, is taken from SEMM [62]. The Software Engineering Methodology Manual (SEMM) "addresses the various aspects of software engineering activities in the Westinghouse Electric Company, Nuclear Fuel Engineering department […]". SEMM is based on the document EP-310 which "presents the requirements that software development must follow." SEMM "presents the recommended best practices to be followed to meet the requirements of EP-310."

SEMM starts with describing different software life cycle models that may be used, for example the waterfall model and the spiral model. Then the different activities that are common to all of the models are treated, such as requirements specification, analysis and design, implementation etc.

The largest part of SEMM is the 21 appendices that are detailed guidelines of for example how to code in different programming languages (coding standards), how to design and maintain a database and how to design a GUI.

The only important reference to unit testing in SEMM is located in Section 2.6 – Software Test Plan Specification under the header "Other Types of Test Cases/Testing", where SEMM states:

"Projects may elect to perform unit testing during their implementation and/or formal testing phases. Whenever these tests are to be conducted during formal testing, the same information provided for version specific tests is to be provided for the unit tests in the Software Test Plan or reference provided to other Quality Records containing the information. Informal unit testing drivers and results should also be captured for possible re-use/review at a later date."

The information that is to be provided for version specific tests is:

- Type of test
- Relationship of test case to specific requirement(s) and/or design aspect(s)
- Description [or] test procedure [or] reference to test procedure
    - Directions or step-by-step instructions

- Inputs
- Expected results
- Method(s) to establish acceptability of the results

Chapter 3 in SEMM is dedicated to the "software inspection process". SEMM recommends that formal inspections are conducted for all phases of the software life cycle, and that "typically, about 15% of the total project resources should be allocated to inspections." However, "no more than 20% of an individual engineer's time should be consumed by inspections."

Personnel participating in software inspections may assume one or more of the following roles:

- "The **Producer** provides the inspection review material describing the work product being inspected."
- "The **Moderator** serves as a Reviewer of the work product, conducts the inspection meeting, keeps it on-track and on-time and is the arbitrator in determining what is required to be changed. In addition, the **Moderator** verifies the resolution of issues which are identified in the inspection."
- "The **Reviewers** examine the work product, identifying potential defects, prior to the inspection meeting, and participate in the inspection meeting."
- "The **Reader** presents the contents of the work product during the inspection meeting. This is typically performed by one of the Reviewers."

The inspection process consists of the following steps:

- **"Inspection Preparation** - the Producer prepares the review package for the Reviewers."
- **"Individual Review** - the Reviewers individually study the review package and identify potential defects in the work product."
- **"Inspection Meeting** - the Producer, Reviewers, and Moderator collectively examine the work product and compile a list of defects to be processed by the Producer."

- **“Inspection Follow-up** - the Producer provides the Moderator documentation of the disposition of the defects identified in the Inspection Meeting, which are verified by the Moderator. The Moderator may then sign off as the verifier of the work product being inspected.”

SEMM also provides references to checklists that are to be used during the inspections.

D Cyclomatic complexity

Cyclomatic complexity [32] is a software metric used to measure the logical complexity of a program [46]. The metric depends on the number of edges and nodes in the flow graph for a particular program. In other words, programs become more complex the more decisions and loops the program contains. Cyclomatic complexity may be calculated with the following formula:

$$\text{Complexity(Graph)} = \text{Edges} - \text{Nodes} + 2$$

Cyclomatic complexity provides a value that is equal to the number of linearly independent paths in a program. As in the case of flow graphs, it is time consuming as well to calculate cyclomatic complexity as to draw flow graphs for every unit. Lindegren [29] describes that if cyclomatic complexity is to be calculated for every function within a project, a tool is needed to do the calculations. Therefore, the calculations are recommended to be automated with help from software tools.

E Flow graph notation

A flow graph is a simple notation for the representation of control flow within a program. “Any procedural design representation can be translated into a flow graph.” [46] An example of a flow graph is provided in Figure E.1. The nodes represent procedural statements while the arrows represent the flow of control. A flow graph should always start from a single node and end with a single node. Flow graphs are useful when designing white-box tests that involve paths and branches.

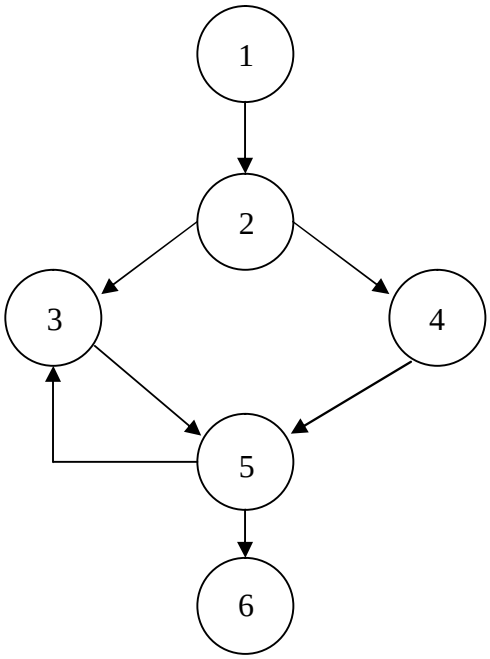


Figure E.1: Flow graph example

F Code for operation LpSänd in class Rodscanner

```
'** FUNCTION ********************************************************
'
'   Namn:         LpSänd
'   Parameterar:  (I) Meddelande att sända
'   Beskrivning:  Länkprotokoll, sändning. Meddelandet läggs i ett
'                 telegram, STX, BCC och ETX adderas.
'   Returnerar:   0 om OK, <0 annars
'
Private Function LpSänd(ByVal meddelande() As Byte) As Integer
   Dim repetition As Integer = 0
   Dim tillstånd As Integer = 1
   Dim returkod As Integer = 99
   Dim kvittens() As Byte

   While returkod = 99
      Select Case tillstånd
         Case 1
            'Om inte för många omsändningar, skicka telegram
            If repetition > MAX_REPETITION Then
               returkod = Me.ReturKod.EjKontakt 'För många
                                    'omsändningar utan svar, avbryt
            Else
               LpSkrivMeddelande(meddelande)
               repetition += 1
               tillstånd = 2
            End If
         Case 2
            'Läs kvittens. Följande kan hända:
            'ACK        färdig
            'timeout    skicka igen
            'annat tkn  antagligen störning, läs igen, vänta på
                       timeout
            Dim ret As Integer = m_Port.ReadWait(1, 0, TIMEOUT,
                                                 kvittens)
            If ret = m_Port.ReturnCode.OK And kvittens(0) = ACK Then
               returkod = Me.ReturKod.OK
            ElseIf ret = m_Port.ReturnCode.OK And kvittens(0) = NAK
            Then
               tillstånd = 1
            ElseIf ret = m_Port.ReturnCode.Timeout Then
               tillstånd = 1
            End If
      End Select
   End While

   Return returkod
  End Function
```

G Unit testing documentation

Only the test cases for the operation LpSänd are shown in the test case documentation, and only the faults found in the operation LpSänd are shown in the fault report.

Test Case Documentation

Unit tested: Rodscanner

Test case ID	Test Item	Description	Input		Result state	Additional information
			Setup	Actions		
	…					
	…					
	…					
	LpSänd					
	Equivalence partitioning/Boundary value analysis	Identified using equivalence partitioning and boundary value analysis. These tests and several others are also covered by the black-box tests for LpSkrivMeddelande				
022	LpSänd	Test with message size 1 and boundary value content hex00	message = [00], set serialIOStub to return OK and ACK	LpSänd(message)	0 returned	

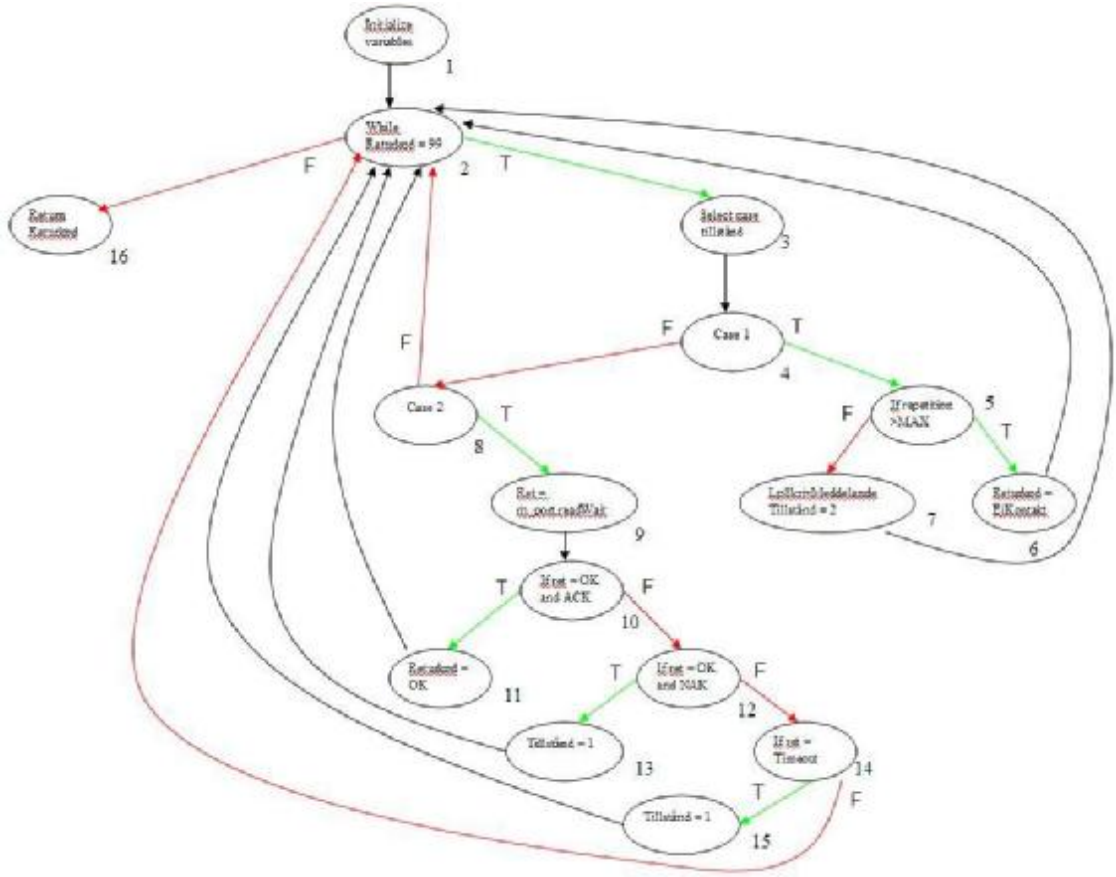
023	LpSänd	Test with message size 20 and boundary value content hex01, hex7F, hex00	message = [20,7F,00,20,7F,00,...], set serialIOStub to return OK and ACK	LpSänd(message)	0 returned	
024	LpSänd	Test if SerialIOstub returns NAK	message = [33], set serialIOStub to return OK and NAK	LpSänd(message)	< 0 returned	
025	LpSänd	Test if SerialIOstub returns Timeout and that LpSänd tries to send message a maximum of MAX_REPETITION times	message = [33], set serialIOStub to return ReturnCode.Timeout	LpSänd(message)	< 0 returned, serialIOstub.m_numberOfWrites = MAX_REPETITION	
	Basis Path testing	Independent paths identified by using a flow graph. Cyclomatic complexity calculated using "Project Analyser" and flow chart generated using "Visustin" (See sheet "Extras LpSänd")				
026	LpSänd	Test independent path 1-2-3-4-5-7-2-…	set serialIOstub to return ReturnCode.OK and ACK	LpSänd(message)	ReturnCode.OK returned	content of message does not matter
027	LpSänd	Test independent path (1)-2-3-4-8-9-10-11-2-…	set serialIOstub to return ReturnCode.OK and ACK	LpSänd(message)	ReturnCode.OK returned	content of message does not matter
028	LpSänd	Test independent path (1)-2-3-4-8-9-10-12-13-2-…	set serialIOstub to return ReturnCode.OK and NAK	LpSänd(message)	ReturnCode.EjKontakt returned	content of message does not matter
029	LpSänd	Test independent path (1)-2-3-4-8-9-10-12-14-15-2-…	set serialIOstub to return ReturnCode.Timeout	LpSänd(message)	ReturnCode.EjKontakt returned	content of message does not matter

030	LpSänd	Test independent path (1)-2-3-4-8-9-10-12-14-2-…	set serialIOstub to first return ReturnCode.NoPortOpen, then ReturnCode.OK and ACK	LpSänd(message)	ReturnCode.OK returned	content of message does not matter
	Loop testing	Identified using Appendix G in the unit testing methodology				
031	LpSänd, while loop	2 iterations (minimum)	set serialIOstub to return ReturnCode.OK and ACK	LpSänd(message)	ReturnCode.OK returned	content of message does not matter
032	LpSänd, while loop	3 iterations	set serialIOstub to first return ReturnCode.Timeout, then ReturnCode.OK and ACK	LpSänd(message)	ReturnCode.OK returned	content of message does not matter
033	LpSänd, while loop	6 iterations	set serialIOstub to first return ReturnCode.Timeout two times, then ReturnCode.OK and ACK	LpSänd(message)	ReturnCode.OK returned	content of message does not matter
034	LpSänd, while loop	7 iterations ("maximum" if only Timeout or NAK is received)	set serialIOstub to return ReturnCode.Timeout	LpSänd(message)	ReturnCode.EjKontakt returned	content of message does not matter
035	LpSänd, while loop	8 iterations	set serialIOstub to first return ReturnCode.NoPortOpen, then ReturnCode.Timeout 3 times	LpSänd(message)	ReturnCode.EjKontakt returned	content of message does not matter

036	LpSänd, while loop	30 iterations	set serialIOstub to first return ReturnCode.NoPortOpen 28 times, then ReturnCode.OK and ACK	LpSänd(message)	ReturnCode.OK returned	content of message does not matter
	Multiple condition testing	Identified using Appendix F in the unit testing methodology. See sheet "Extras LpSänd" for details				
037	LpSänd, Node 10	true/true	set serialIOstub to return ReturnCode.OK and ACK	LpSänd(message)	ReturnCode.OK returned	content of message does not matter
038	LpSänd, Node 10 and 12	node 10: true/false, node 12: true/true	set serialIOstub to return ReturnCode.OK and NAK	LpSänd(message)	ReturnCode.EjKontakt returned	content of message does not matter
039	LpSänd, Node 10 and 12	Node 10: false/true, Node 12: false/false	set serialIOstub to return ReturnCode.Timeout and ACK	LpSänd(message)	ReturnCode.EjKontakt returned	content of message does not matter
040	LpSänd, Node 10 and 12	Node 10: false/false, Node 12: false/true	set serialIOstub to return ReturnCode.Timeout and NAK	LpSänd(message)	ReturnCode.EjKontakt returned	content of message does not matter
041	LpSänd, Node 12	true/false	set serialIOstub to first return ReturnCode.OK and 5 (NAK = 6), then return ReturnCode.OK and ACK	LpSänd(message)	ReturnCode.OK returned	content of message does not matter
	Data flow testing	Two paths were identified by using guidelines for data flow testing				

042	LpSänd	Test path 1-2-3-4-5-7-2-3-4--8-9-10-12-14-15-2-3-4-5-7-2-3-4-8-9-10-12-13-2-3-4-5-7-2-3-4-8-9-10-11-2-16	set serialIOstub to first return ReturnCode.Timeout, then return ReturnCode.OK and NAK, and finally return ReturnCode.OK and ACK	LpSänd(message)	ReturnCode.OK returned	content of message does not matter
043	LpSänd	Test path 1- (2-3-4-5-7-2-3-4-8-9-10-12-13)x4 -2-3-4-5-6-2-16	set serialIOstub to return ReturnCode.OK and NAK	LpSänd(message)	ReturnCode.EjKontakt returned	content of message does not matter
	…					
	…					
	…					

Table G.1: Test case documentation for LpSänd



Flow Graph for LpSänd

T = TRUE
F = FALSE

Cyclomatic complexity: 8

Independent paths:

1-2-16*	Not possible
1-2-3-4-5-6-2-…	Not possible
1-2-3-4-5-7-2-…	Test Case ID: 026
1-2-3-4-8-2-…	Not possible
(1)-2-3-4-8-9-10-11-2-16*	Test Case ID: 027
(1)-2-3-4-8-9-10-12-13-2-…*	Test Case ID: 028
(1)-2-3-4-8-9-10-12-14-15-2-…*	Test Case ID: 029
(1)-2-3-4-8-9-10-12-14-2-…*	Test Case ID: 030

* = These paths can not be executed directly. They must be exercised as part of a longer path (that is, as a subpath). For instance, to reach node 8, node 5 must have been evaluated to false at least once in the past

(1) means that node 1 can not be in such a subpath. Node 1 will be excluded in such cases

Multiple condition Testing

Node 10

ret = OK	ACK	Tested	Test case ID
T	T	X	037
T	F	X	038
F	T	X	039
F	F	X	040

Node 12

ret = OK	NAK	Tested	Test case ID
T	T	X	Redundant, covered by ID038
T	F	X	041
F	T	X	Redundant, covered by ID040
F	F	X	Redundant, covered by ID039

Table G.2: Extra Excel sheet for LpSänd

Fault Report

Classifications

Impact classes	1 = Critical	The fault results, or may result, in failures that stop the software system, a software subsystem or a software unit from operating
	2 = Major	The fault results, or may result, in failures that cause an incorrect, incomplete or inconsistent output
	3 = Minor	The fault does not result in failures that cause an incorrect output
Type classes	1 = Calculation/Logic	Faults concerning for example the correctness of algorithms, predicate expressions or error handling
	2 = Data handling	Faults concerning for example initialization and assignment of data or the use of correct data types
	3 = Interface	Faults concerning for example arguments of class operations or overall functionality of class operations
	4 = Support code	Faults concerning for example test drivers or stubs
	5 = Design/Specification	Faults concerning for example return types of class operations or wrong number of states of a class. Although similar to the faults in the 'Interface' category, observe that in the 'Design/specification' category the faults are in the design or in the specification of the unit
	6 = Other	Faults that either seems to belong to more than one of the other categories, or does not fit into any of the other categories

Fault ID	Situation of occurence *The situation when the failure(s) occur(s)*	Effect of the fault *The effects of the fault (e.g. observed failures)*	Fault description	Impact	Type	How the fault was found *For example, the test case(s) that lead to the discovery of the fault.*	Date the fault was found	If the fault has been corrected *Yes or No*
	…							
005	In method LpSänd, when trying to send the message a maximum number of times	LpSänd tries to send the message MAX_REPETITION + 1 times	If repetition > MAX_REPETITION, should be If repetition = MAX_REPETITION	3	1	Inspecting code when designing data flow test cases. Confirmed by test case ID 025 (redesigned to find this fault), and also found by test case ID 035	2005-11-29	No

006	In method LpSänd	If LpSkrivMeddelande should return a value, it can be taken care of and when tillstånd = 1, if sending message fails, no need to enter tillstånd 2	Dont handle possible return value from LpskrivMeddelande	3	1	Inspection when designing white-box test cases	2005-11-29	No
	...							
	...							
	...							

Table G.3: Fault report for LpSänd

Unit Test Report

		GUIDANCE
Unit Description	Rodscanner	*Description (e.g. name) of the unit the test report concerns. Should include version number if applicable*
Last date unit test cases where run	05-12-12	*When the unit was last tested*
Have the test goals for the unit been reached	Yes	*Yes or No*
Test level goals for the unit	At least branch coverage	*How much testing was planned to be done for the unit, e.g. what level of coverage was to be achieved?*
Test level reached for the unit	Coverage level higher than branch coverage reached for all operations. Different coverage level reached for different operations.	*What level of testing was achieved for the unit, e.g. what level of coverage was achieved?*
Reference to associated test plan(s) and test cases	No test plan exists. Test cases documented in UnitTestRodscanner.xls. Test cases implemented in RodscannerTest.vb.	*References to the test plan and the test cases (e.g. files) concerning the unit*
Total number of test cases for the unit	191	*Total number of test cases for the unit.*
System used	Pentium 1.7GHz, 512 RAM, OS: WinXP Professional SP1	*The computer system and operating system used when the tests were last performed*

List of faults that have not been corrected yet	**Fault ID**	**Reason that fault is not corrected**
	001-028	Decision to correct faults is the responsibility of Supervisor

Table G.4: Unit test report for LpSänd

H Test case implementation

The following code is a simplified version of the class RodscannerTest, which only contains the information relevant for the operation LpSänd. Only one test case driver is shown for each of the unit testing methods, e.g. only the test case ID029 is shown for the basis path test cases. The test case drivers are implemented for use with NUnit. The comments are in Swedish according to the programming rules defined for the system containing the Rodscanner class. The test case drivers use both the SerialIO class stub and the PrivateHelper class.

```
'** CLASS
'*****************************************************************
'
'   Namn:
'   Skapad av:     Fredrik Strandberg  & Stefan Lindberg
'
'   COPYRIGHT Westinghouse AB
'
'   Beskrivning:      Innehåller testfall för klassen Rodscanner,
'                     implementerade med attribut för NUnit
'
'   Ändringshistorik:
'   $Log:
'
Imports NUnit.Framework
Imports WE.PRS.Fasader

<TestFixture()> _
Public Class RodscannerTest

#Region "Constants"

   Private Const LPSÄND As String = "LpSänd"
   Private Const ACK As String = "ACK"
   Private Const NAK As String = "NAK"

#End Region

#Region "Private fields"

   Private m_rodscanner As WE.PRS.HANDLER.Rodscanner
   Private m_meddelande() As Byte
   Private m_serialIOstub As SerialIo

#End Region

#Region "Setup"

   ' FUNCTION *************************************************
   '
   '   Name:         setup
```

```
    '   Description:  Denna metod körs innan varje testfall
    '
    <SetUp()> _
    Public Sub setup()

        m_rodscanner = New WE.PRS.HANDLER.Rodscanner
        m_serialIOstub = New SerialIo
        PrivateHelper.setInstanceField("m_Port", m_rodscanner,
                                               m_serialIOstub)

    End Sub

#End Region

#Region "Test cases"


#Region "LpSänd"

#Region "Equivalence/Boundary test cases"

    <Test(), Category(LPSÄND)> _
    Public Sub testCaseID022()

        m_meddelande = New Byte() {&H0}
        m_serialIOstub.m_codeToReturn = SerialIo.ReturnCode.OK
        m_serialIOstub.m_readWaitBuffer = New Byte()
             {PrivateHelper.getStaticField(m_rodscanner.GetType, ACK)}

        Assert.AreEqual(0, PrivateHelper.runInstanceMethod(LPSÄND,
             m_rodscanner, m_meddelande))

    End Sub

#End Region

#Region "Basis path test cases"

    <Test(), Category(LPSÄND)> _
    Public Sub testCaseID029()

        'innehållet i meddelande spelar ingen roll
        m_meddelande = New Byte() {&H33}

        m_serialIOstub.m_codeToReturn = SerialIo.ReturnCode.Timeout

        Assert.AreEqual(CType(WE.PRS.HANDLER.Rodscanner.ReturKod.EjKontakt,
             Integer), _
        PrivateHelper.runInstanceMethod(LPSÄND, m_rodscanner, m_meddelande))

    End Sub

#End Region

#Region "Loop test cases"

    <Test(), Category(LPSÄND)> _
    Public Sub testCaseID033()

        'innehållet i meddelande spelar ingen roll
        m_meddelande = New Byte() {&H33}
```

```
        m_serialIOstub.m_returnSeveralCodes = True
        m_serialIOstub.m_codesToReturn = New SerialIo.ReturnCode()
                                    {SerialIo.ReturnCode.Timeout, _
                                     SerialIo.ReturnCode.Timeout, _
                                     SerialIo.ReturnCode.OK}

        m_serialIOstub.m_readWaitBuffer = New Byte()
        {PrivateHelper.getStaticField(m_rodscanner.GetType, ACK)}

        Assert.AreEqual(CType(WE.PRS.HANDLER.Rodscanner.ReturKod.OK,
             Integer), _
        PrivateHelper.runInstanceMethod(LPSÄND, m_rodscanner, m_meddelande))

    End Sub

#End Region

#Region "Multiple condition test cases"

    <Test(), Category(LPSÄND)> _
    Public Sub testCaseID039()

        'innehållet i meddelande spelar ingen roll
        m_meddelande = New Byte() {&H33}

        m_serialIOstub.m_codeToReturn = SerialIo.ReturnCode.Timeout
        m_serialIOstub.m_readWaitBuffer = New Byte()
        {PrivateHelper.getStaticField(m_rodscanner.GetType, ACK)}

        Assert.AreEqual(CType(WE.PRS.HANDLER.Rodscanner.ReturKod.EjKontakt,
             Integer), _
        PrivateHelper.runInstanceMethod(LPSÄND, m_rodscanner, m_meddelande))

    End Sub

#End Region

#Region "Data flow test cases"

    <Test(), Category(LPSÄND)> _
    Public Sub testCaseID042()

        'innehållet i meddelande spelar ingen roll
        m_meddelande = New Byte() {&H33}

        m_serialIOstub.m_returnSeveralCodes = True
        m_serialIOstub.m_codesToReturn = New SerialIo.ReturnCode()
                                    {SerialIo.ReturnCode.Timeout, _
                                     SerialIo.ReturnCode.OK, _
                                     SerialIo.ReturnCode.OK}

        m_serialIOstub.m_returnSeveralBytes = True
        m_serialIOstub.m_bytesToReturn = New Byte()
             {PrivateHelper.getStaticField(m_rodscanner.GetType, NAK), _
              PrivateHelper.getStaticField(m_rodscanner.GetType, NAK), _
              PrivateHelper.getStaticField(m_rodscanner.GetType, ACK)}

        Assert.AreEqual(CType(SerialIo.ReturnCode.OK, Integer), _
        PrivateHelper.runInstanceMethod(LPSÄND, m_rodscanner, m_meddelande))
```

```
    End Sub
#End Region
#End Region
#End Region
End Class
```

I Guidelines

I.1 Guidelines for data flow testing

The following guidelines, proposed by Binder [11], are most effective on operations that have a cyclomatic complexity value of maximum ten, and preferably six or fewer variables:

1. Prepare a control flow graph for the operation under test.
2. Tabulate all D (define), U (use), and K (termination) action by variable, by node. See Table I.1 below for an example.
3. Select a color for each variable. Use erasable ink on an easily erased sheet – water–soluble markers on presentation transparencies work well. Erasing (step 5) can be done with a damp cloth.
4. For each variable:
    a. Use the color for the current instance variable.
    b. Draw a solid circle at each node with a D or K action.
    c. Draw an open circle at each node with a U action.
    d. For each D action:
        i. For each U action:
        Look for a definition-clear subpath from the D to the U action (there may not be one). Trace this path in the current variable's color. Try to keep the line near the control flow path.
        End for
    End for
End for
5. By inspection, choose an entry-exit path with lots of DU paths.
    a. Add this entry-exit path to your test suite.
    b. Erase all DU subpaths (drawn in step 4) along this entry-exit path.
    c. Repeat step 5 until there are no more DU paths to erase.
6. Determine values of variables and arguments that are required to exercise the test paths. Discard infeasible paths.

I.2 Guidelines for loop testing

The following guidelines are proposed when testing different loop constructs (if iterators are used instead of loops, replace loop in this section with iterator):

- *Simple loops [46].*

  Consider a loop that may pass ***n*** times.

  1. Skip the loop entirely.
  2. Only one pass through the loop.
  3. Two passes through the loop.
  4. m passes through the loop where m < n
  5. n – 1, n, n + 1 passes through the loop.

- *Nested loops [11].*

  1. Find the shortest or simplest path to the loop. The input and state required for this path (up to the loop) will be required for each of the loop tests.
  2. Test focus begins at the innermost loop and proceeds to the outermost loop.
  3. For each variable-iteration focus loop:
     a. For each loop control case (min, min + 1, typical, max – 1, max):
        i. Determine the input/state values needed to drive focus loop control to the control case.
        ii. Determine the input/state values needed to drive nonfocus loop controls to the nonfocus cases of (minimum, typical, or maximum).
        iii. Add excluded values and out-of-range checks if indicated (e.g. max + 1 and min - 1).

     End for

     b. Add the loop conditions to the partial path conditions – you may have to choose a different path to the loop to achieve the loop to control conditions (other than the path determined in step 1).

  End for

  4. Set up a test that will run all loops to maximal control case (try and design as many test cases as possible that include min, min + 1, typical, max – 1 and max for all loops).

- *Concatenated loops.*

  If each of the concatenated loops is independent the same approach may be used as for simple loops. If the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

- *Unstructured loops.*

  Unstructured loops should not be tested but redesigned.

J Definitions and abbreviations

J.1 Definitions

.NET

In this thesis, .NET will refer to the .NET framework. The .NET framework is an environment in which you can build, create and deploy applications, and includes a set of framework classes. The .NET framework also includes the Common Language Runtime which provides a Common Type System, garbage collection and execution support.

Black-box testing

Testing performed without knowledge of the inner structure of the unit being tested. Testing is performed to ensure that the specification of the unit being tested is fulfilled.

Debugging

The process of locating and correcting faults.

Error

A human action that produces a fault.

Fault

A missing or incorrect specification, design or code.

Failure

The manifested inability of a system or component to perform a required function within specific limits.

Method

A systematic way of performing tasks.

Methodology

A documented and organized system of methods.

Postcondition

A condition that is satisfied after an operation has been called.

Precondition

A condition that must be satisfied before an operation is called.

Safety

A judgement of how likely it is that the system will cause damage to people or its environment.

Safety-critical systems

A system whose failure may result in injury, loss of life or major environmental damage.

Software engineering

Definition by IEEE [24]:

(1) Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

(2) Software engineering is the study of approaches as in (1).

Software quality

See Appendix A.1.

Software quality assurance

The establishment of a framework of organizational procedures and standards which lead to high-quality software.

Software quality control

The definition and enactment of processes which ensure that the project quality procedures and standards are followed by the software development team.

Software testing

An activity concerned with uncovering evidence of faults in software and also with ensuring that the product meets the requirements from the customers and the users.

Structured testing process

Formal standards, procedures or guidelines are being used when testing of the software is performed.

Test case

A set of conditions or variables under which a tester will determine if a requirement upon an application is partially or fully satisfied [64].

Test coverage

Test coverage is a measure of how completely a test case or test cases exercise the capabilities of a piece of software.

Test suite

A collection of test classes and/or test cases.

Test-to-fail

Testing with invalid values, e.g. testing with invalid or illegal inputs.

Test-to-pass

Testing with valid values.

Unit

A unit may be a function/procedure or a data collection. In object-oriented development a unit is a class or an interdependent cluster of classes.

Unit testing

The testing of a single unit, separated from other units of the program.

Unstructured testing process

No formal standards, procedures or guidelines are being used when testing of the software is performed.

Validation and verification

The processes of checking and analyzing whether a software product conforms to its specification and meets the needs of the users.

White-box testing

Testing performed with knowledge of the inner structure of the unit being tested.

J.2 Abbreviations

CUT	Class under test
EP-310	Engineering Procedure 310
IEEE	Institute of Electrical and Electronics Engineers
IDE	Integrated Development Environment
LSP	Liskovs Substitution Principle
ManIT	Manufacturing-IT
ODC	Orthogonal defect classification
PLC	Programmable Logic Controller
QMS	Quality Management System
SEMM	Software Engineering Methodology Manual
UTM	Unit testing methodology