

Lappeenranta University of Technology  
Faculty of Technology Management  
Department of Information Technology

## **A proposal for a repertoire of software testing methods**

Examiners: Professor Kari Smolander  
D.Sc.(Tech) Ossi Taipale

Instructor: D.Sc.(Tech) Ossi Taipale

Lappeenranta, 10.05.2010

Petr Bavin  
Kalliopellonkatu 10 A 1  
53850 Lappeenranta  
Petr.Bavin@lut.fi  
+358-466155544

## **ABSTRACT**

Lappeenranta University of Technology  
Faculty of Technology Management  
Department of Information Technology

Petr Bavin

### **A proposal for a repertoire of software testing methods.**

Master's Thesis

2010

75 pages, 24 figures, 9 tables

Examiners: Professor Kari Smolander  
D.Sc.(Tech) Ossi Taipale

Keywords: software testing, software testing methods, software quality estimation

Nowadays software testing and quality assurance have a great value in software development process. Software testing does not mean a concrete discipline, it is the process of validation and verification that starts from the idea of future product and finishes at the end of product's maintenance. The importance of software testing methods and tools that can be applied on different testing phases is highly stressed in industry.

The initial objectives for this thesis were to provide a sufficient literature review on different testing phases and for each of the phases define the method that can be effectively used for improving software's quality. Software testing phases, chosen for study are: unit testing, integration testing, functional testing, system testing, acceptance testing and usability testing.

The research showed that there are many software testing methods that can be applied at different phases and in the most of the cases the choice of the method should be done depending on software type and its specification. In the thesis the problem, concerned to each of the phases was identified; the method that can help in eliminating this problem was suggested and particularly described.

## **FOREWORD**

This master's thesis was written as part of MASTO project at Lappeenranta University of Technology during a six-month period from November 2009 to May 2010. I would like to thank everyone who had helped me during this period.

I would like to thank my instructors Dr. Ossi Taipale and Prof. Kari Smolander for all support and advices I have received during writing this thesis. I owe special thanks for providing opportunity to make my own contribution in developing new software testing standard ISO 29119. This work was really interesting for me.

I would like to thank my family – people who always believe in me and invigorate me to go forward. I would like to say special thanks to my friend Vadim Matveev, who has helped me a lot.

Lappeenranta, 10.05.10

Petr Bavin

## Table of Contents

1. INTRODUCTION .....	5
1.1 Background .....	5
1.2 Objectives and restrictions .....	6
1.3 Structure of the thesis .....	6
2. SOFTWARE TESTING .....	7
2.1 Basic concepts of software testing.....	7
2.2 Nature of Testing and Testing Process .....	8
2.3 Testing Principles.....	9
2.4 Types of errors .....	11
2.5 Human software testing.....	12
2.6 Automated Software Testing.....	16
3. SOFTWARE TESTING PHASES .....	17
3.1 Basic concepts of software testing phases.....	17
3.2 Unit testing .....	20
3.3 Integration testing .....	22
3.4 Function testing.....	25
3.5 System testing .....	30
3.6 Acceptance testing .....	32
3.7 Usability testing .....	34
4. SUGGESTION FOR A REPERTOIRE OF TESTING METHODS .....	36
4.1 Unit Testing.....	36
4.2 Integration testing .....	43
4.3 Functional testing.....	53
4.4 System testing .....	57
4.5 Acceptance testing .....	63
4.6 Usability testing .....	67
5. RESULTS AND DISCUSSION .....	70

5.1 Discussion on suggested methods.....	70
6. CONCLUSION .....	73
REFERENCES .....	74

## **SYMBOLS AND ABBREVIATIONS**

ANSI	American National Standards Institute
ASTM	American Society for Testing and Materials
CDE	Coordination Development Environment
CFT	Call for Testing
CP-net	Coloured Petri Net
IEEE	Institute of Electrical & Electronics Engineers
ISO	International Standards Organization
IEC	International Electrotechnical Commission
IT	Information Technology
HTML	Hyper Text Markup Language
LOC	Lines of code
OO	Object-Oriented
QA	Quality Assurance
RAD	Rapid Application Development
SDT	Software Development Technologies
TDD	Test Driven Development
TSMC	Testing Sequence of Message Calls
UAT	User Acceptance Testing
UCD	User-Centric Design
UI	User Interface
UML	Unified Modeling Language

XML                    Extensible Markup Language  
XP                      eXtreme Programming

# **1. INTRODUCTION**

## **1.1 Background**

Nowadays software testing and quality assurance have a great value in software development process. The role of a testing is significant both for small projects and for complex systems, irrespective of the project's budget and the number of people involved in it. The reason is that a testing is an activity which focuses on the quality of a software and in many respects determines its success.

Software testing does not mean concrete discipline, it is the process of validation and verification that starts from the idea of future product and finishes at the end of product's maintenance. It is a complex process, which consist of many separate phases. Some of these phases are common, such as unit testing, during which the program is divided on modules and then modules are tested separately; or functional testing, main purpose of which is to ensure that software's functionality met functional specification. The presence of certain phases depends on the type of software, e.g. security testing, where secure abilities of the software are checked; or stress testing, which ensures that program is able to work with heavy load.

The base of software testing is two entities: Methods and Tools. Testing methods are techniques that help to find errors in the program and improve its quality. Methods describe how test cases should be developed; specify what kind of data should be used; set the criteria for passing or failing test cases. Testing tools are software instruments that are used for detecting errors in the program. Tools usually use methods for performing automated testing. Comparing to human testing, automated testing allows detecting errors more quickly with less quantity of human resources. It is very important for contemporary software projects, where people and the budget are the bottlenecks.



## **1.2 Objectives and restrictions**

The initial objectives for this thesis were to provide a sufficient literature review on different testing phases and for each of the phases define the method that can be effectively used for improving software's quality.

The choice of testing phases was founded on SDT Dotted U-Model, which was proposed by Kit (Kit, 1996). This model describes testing phases for the whole cycle of software development process and defines relations between them. SDT Dotted U-Model helps to organize manageable testing process, minimize risks and make testing more effective. Software testing phases, chosen for study are: unit testing, integration testing, functional testing, system testing, acceptance testing and usability testing.

The thesis was done as a part of a software testing and quality assurance research project MASTO at Lappeenranta University of Technology. This thesis also made contribution for "Study Group for Tools and Methods of Software Testing (ISO/IEC JTC1/SC7 WG4)" – international standardization group, that makes researches in the area of software testing tools and methods. Methods discussed in thesis were suggested to the standardization group as methods to be included in new software testing standard ISO 29119.

## **1.3 Structure of the thesis**

Section 2 of thesis is a survey of software testing discipline: principles, testing methods and tools. Section 3 presents information about defined software testing phases and discusses how do they impact on the quality of software and what basic techniques are used during appropriate phase. Section 4 provides a suggestion for a repertoire of testing methods to be used on defined phases for software's quality improvement. Section 5 provides the discussion of methods, which were chosen as a suggestion in section 4.

## 2. SOFTWARE TESTING

The term software testing has different definitions in the literature. Myers defines software testing as a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended (Myers, 2004).

IEEE standard 610.12-1990 (1990) defines software testing as:

- The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component.
- The process of analyzing a software item to detect the difference between existing and required conditions and to evaluate the features of the software items.

In the definitions it is mentioned that software testing is a process of applying input conditions to software product and inspecting obtained results. Besides, there are differences between software testing and other software development phases.

Basically, software testing answers two questions: does one develop right product (validation) and does one develop product right (verification) (Kit, 1996). This means that software testing covers the whole life cycle from product development to maintenance. The processes of software validation and verification i.e. software testing are present in any phase of software development cycle.

This chapter contains definitions of basic concepts of software testing such as test, test case, testing process, testing principles and types of errors. The value of automated software testing is also explained.

### 2.1 Basic concepts of software testing

The definitions of the terms are quoted, from standards and related literature.

**Test** is defined as an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component (IEEE/ANSI 610.12-1990 standard, 1990).

**Test case** is a set of inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specified requirement (IEEE/ANSI 610.12-1990 standard, 1990).

Testing strategy is used in selecting test cases that implant the methods and tools. According to Myers two of the most prevalent strategies include black-box testing and white-box testing (Myers, 2004).

**Black-box** (or data driven) testing strategy offers to take a look at program, like at a black box and the goal of tester is to find circumstances, where the program does not work as it is supposed to work. While using this approach, test data are derived solely from the specifications (i.e., without taking advantage of knowledge of the internal structure of the program) (Myers, 2004). The main disadvantage of this strategy is that it does not take into account the structure of the program, so it does not cover all different ways of program behavior.

**White-box** (or logic driven) testing strategy offers to analyze internal structure of the program. This strategy derives test data from an examination of the program's logic (and often, unfortunately, at the neglect of the specification) (Myers, 2004). The main disadvantage of this strategy is that it can be very difficult to apply this strategy, when the program size is large.

## **2.2 Nature of Testing and Testing Process**

Software testing differs from other activities of software development process because software testing is destructive process. That means that the main aim of a tester is to find weaknesses of the software – to find as many errors as it is possible. A successful test case finds errors in the program and unsuccessful does not. The job of a software tester is rather specific and not all the people can be talented testers, because human nature mostly is constructive than destructive. “Testing is a positive and creative effort of destruction. It takes imagination, persistence and a strong sense of mission to systematically locate the weaknesses in a complex structure and to demonstrate its failures” (Kit, 1996, pg . 22).

It is also hard to test own code. The person, who has implemented the code starts testing from the point of view that everything works properly. It is hard to think that your own implementation is error prone. That is why developers and testers are usually separate groups and sometimes cannot find mutual understanding. In spite of this “everybody – testers, marketing people, and managers – needs to understand that testers are adding value to the product by discovering errors and getting them on the table as early as possible” (Kit, 1996, pg. 23).

Kit mentions that, testing can be separated into two basic form – validation and verification and these definitions concern not only to code testing, but also testing documents, requirements and other non-executable forms.

The definitions of validation and verification provided by IEEE/ANSI, 1990 [Std 610.12-1990]:

**Verification** is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

**Validation** is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

Validation helps to answer the question does one develop right product and verification - does one develop product right. According to Kit, “Testing = verification + validation”. Testing is not an activity at one phase of software development, but it is a process that is accomplished through all the life cycles of the software.

### **2.3 Testing Principles**

Although software testing is a technical task, it is always made by people, so human and economic aspects are very important. Myers (Myers, 2004) defines 10 principles of software testing in his book. The principles are listed in Table 2.3.1.

Table 2.3.1 “Vital Program Testing Guidelines” – principles of software testing, Myers (Myers, 2004)

Principle number	Principle
1	A necessary part of a test case is a definition of the expected output or result.
2	A programmer should avoid attempting to test his or her own program.
3	A programming organization should not test its own programs.
4	Thoroughly inspect the results of each test.
5	Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
6	Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
7	Avoid throwaway test cases unless the program is truly a throwaway program.
8	Do not plan a testing effort under the tacit assumption that no errors will be found.
9	The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
10	Testing is an extremely creative and intellectually challenging task.

On the one hand, these principles are obvious and understandable; on the other hand they reproduce the most valuable aspects of software testing. Myers explains that testing should not be accomplished by organization or individual (principles 2, 3), who has developed product or code, because it is hard for author to test his own work and sometimes it is hard to acknowledge that something went wrong. Another important point is that expected result of a test case should be predefined, because often people tend to see what they want to see (principle 1), test cases should be developed both for expected and unexpected input data (principles 5, 6) and the result of a test case should be always inspected (principles 4, 6). Myers also deals with human psychology when mentioning that throwaway tests should be done only when “program is truly throwaway program” (people often tend to hurry and accomplish next step, before doing previous), one does not need to develop test cases under assumption that everything in the program works properly, because it is easy way to get a mistake in an unexpected place (principles 7, 8).

In this set of principles Myers also mentions that “probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section” (Myers, 2004, pg.15) - if the section of the code contained errors, it’s a bottleneck and tester should keep it in mind.

The last principle tells that testing is really difficult and challenging task, but at the same time it is interesting, creative and entirely based on knowledge, experience and imagination of the testers.

## **2.4 Types of errors**

Testing of software is a destructive process. The main task of each tester is to find as many errors in the program, as it is possible. Errors can be classified into groups, for example, depending on the time they occur (runtime errors), syntax and semantic errors.

IEEE/ANSI, 1990 [Std 610.12-1990] provides several definitions of general errors:

- **Mistake**: a human action that produces an incorrect result.

- **Fault**: an incorrect step, process, or data definition in a computer program. The outgrowth of the mistake.
- **Failure**: an incorrect result. The result of the fault.
- **Error**: the amount by which the result is incorrect.

This list describes the way how failures and errors occur in a computer program. The “base” for each error is a mistake, made by humans (as coding is human work). The reason for human mistakes usually lie in human’s nature – stress, tiredness or any factor that does not let person to concentrate. A fault is the result of this error and is the reason of a failure or an error that occurs after.

## **2.5 Human software testing**

This section provides information about tree most popular methods of human testing that can be used during a software development cycle: code inspections, walkthroughs and desk checking. This information will be necessary in sections 3 and 4, where software testing phases and methods that can be applied during these phases are discussed.

The thesis observes software testing as the process of validation and verification activities, as was suggested by Kit (Kit, 1996). Generally, validation helps to answer the question does one develop right product and verification - does one develop product right.

“Software testing and software development are closely related because, for example, approaches, methods, tools, technologies, processes, knowledge, and automation of software development affect testing and vice versa” (Taipale, 2007, pg. 16). Testing as a process of validation and verification affects on each of the phases of software development.

Software testing method can be defined as a definitive procedure, that produce test result (Form and Style for ASTM standards, 2009). Software testing methods has several

classifications: they can be classified by phases, where they can be applied, by complexity of implementation, by efficiency and etc.

There are defined methods of human testing, which are basic and often applied in software projects. According to Myers, code inspections and walkthroughs are the two primary human testing methods, experience with these methods found that they are effective in finding from 30 to 70 of the logic design and coding errors in the typical programs (Myers, 2004). Both methods involve visual inspection, which is performed by the team of participants.

### **Code inspections**

According to Fagan, a code inspection is a set of procedures and error-detection techniques for group code reading (Fagan, 1976). The team, which performs code inspection, consists of the people with roles, presented in the table 2.5.1.

Table 2.5.1. Roles in inspection process.

Role name	Role description	Responsibilities
Moderator	Qualified programmer, but not author of the program.	Distributing materials for inspection session
Programmer	Author of the program	Explaining the logic structure of the program
Tester	Qualified tester	Analyzing logic structure, finding errors

The process of code inspections consist of the following steps:

1. The materials (program listing, design specification, error checklist) are distributed by moderator.
2. Programmer explains the internal structure of the program in details (from statement to statement).



3. The program is analyzed according to the most common programming errors, presented in the checklist.

Moderator checks that participants focus on finding errors, but not on correcting them. The result of code inspections is error list, categorized by types of the errors. It is provided to the programmer for improving the program.

The inspection process is “a way of identifying early the most error-prone sections of the program” (Myers, 2004, pg. 22). Another profit of code inspections is that errors are searched and detected not only by author of code, but also by other qualified participants. That allows to review the code from different points of view and to make the inspection more impartial.

### **Walkthroughs**

Walkthroughs are similar to the code inspections in respect of participants and group work, but procedures of walkthrough are different. Table 2.5.2 provides information about team member, that can be included in the walkthrough process, according to Myers (Myers, 2004).

Table 2.5.2 Walkthrough’s participants.

Participant	Responsibilities
Programmer (author)	Evaluating test cases
Programming-language expert	Evaluating test cases
New programmer	Evaluating test cases; Providing fresh outlook
Someone from other project	Evaluating test cases
Someone from the same programming team	Evaluating test cases
Tester	Evaluating test cases

All the participants are provided the materials as well as in code inspection case, but the procedure of walkthrough is different: one person, who was chosen as tester, brings the set of test cases, which are evaluated by participants during the walkthrough. Errors are detected mostly during the discussion of code constructs between participants.

### **Desk Checking**

Desk checking is also very widely used software testing method, which is simply based on analyzing the code and detecting errors by the programmer himself. It can be viewed as a one-person inspection or walkthrough (Myers, 2004), when person (usually author of the code) checks code with respect to error checklist or walks test data through it.

The main profit of this method is that it can be performed faster than walkthroughs and inspections, as it requires involving only one person, who is familiar with the code. Desk checking is suitable for searching simple errors on the first step of error's detection.

On the other hand this method has several lacks:

- it is not satisfied by the principles of software testing, presented in chapter 2 (programmer should not test his own code);
- method is uncontrolled ;
- a team spirit is also absent.

These lacks lead to inefficient testing, and make this method applicable only on the initial state of code analysis.

## **2.6 Automated Software Testing**

Automated software testing offers automated tools for testing software. The value of automated testing is high, because it helps to accomplish testing activities faster and more effectively rather than human testing. Benefits of automated testing include, that earlier gained information about found errors can be utilized, it allows to start earlier fixing them and to get earlier correct result. It is important, because while developing software product one should keep in mind time and budget aspects and using automated tools that help to accelerate the process.

The popularity of automated software testing is also excited by applying rapid application development. According to Dustin, “the growth of automated test capability has stemmed in large part from growing popularity of rapid application development (RAD), a software development methodology that focuses on minimizing the development schedule while providing frequent, incremental software builds” (Dustin, 2000, pg. XVI). The main purpose of RAD is to satisfy customers as early as possible. For doing this it is necessary to understand the requirements, to fulfill them and then check how they were fulfilled. To accomplish the most activities, which are concerned with software testing one should use automated tools, because often manual testing is labor-intensive and error-prone (Dustin, 2000) and it simply can not provide the test quality of automated testing, especially when taking into account project schedule.

At the moment there are many open source tools that provide automation of verification and validation processes. That means that software teams are able not only use automated tools for free of charge (this point is very important, because one should always keep in mind budget aspects), but also make changes if it is necessary.

### **3. SOFTWARE TESTING PHASES**

In this section the literature review of software testing phases is provided. The choice of testing phases was founded on SDT Dotted U-Model, which was proposed by Kit (Kit, 1996). Software testing phases, chosen for study are: unit testing, integration testing, functional testing, system testing, acceptance testing and usability testing. This section provides general concepts for defined testing phases, describes the importance each of them and provides information about basic methods that are used during these phases.

#### **3.1 Basic concepts of software testing phases**

Software testing phases are activities of validation and verification process. According to Kit, validation activities can be divided into low-level testing and high-level testing (Kit, 1996), depending on what parts of a software product are tested.

Low-level testing performs testing of individual components and requires the knowledge of internal program's structure, usually low-level testing is accomplished by development team. Low-level testing consists of unit testing and integration testing.

High-level testing performs testing of the complete product and is accomplished by testing team, which can be located as inside the same organization, as outside it (another organization that performs testing facilities). It consists of usability testing, function testing, system testing and acceptance testing. High-level testing involves testing whole, complete products (Kit, 1996). More detailed information about low-level and high-level testing activities is provided below.

Figure 3.1.1 presents information about software testing phases, chosen for study. Figure is based on SDT Dotted-U Model (Software Development Technologies, 2010), which was proposed by Kit. This model describes testing phases for the whole cycle of software

development process and defines relations between them. SDT Dotted-U Model helps to organize manageable testing process, minimize risks and make testing more effective.

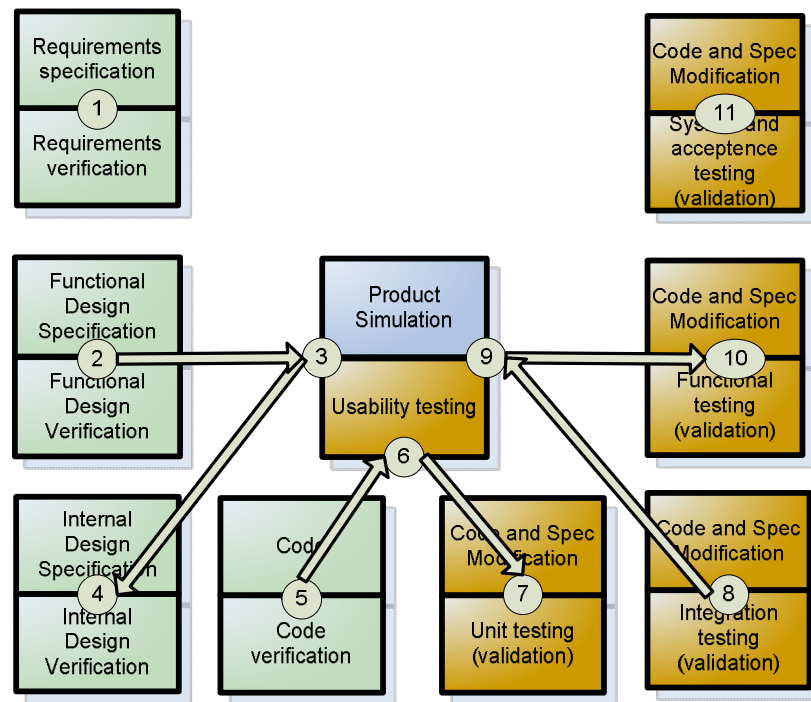


Figure 3.1.1 The SDT Dotted-U Model (Kit, Software Development Technologies, 2010)

The process of verification and validation, defined in SDT U-Dotted Model contains 11 steps:

1. Requirements specification and verification: requirement management is a systematic approach to eliciting, organizing, and documenting the requirements of the system (Dastin, 2000) and requirement phase is separated from other phases of software development.
2. Functional design specification and verification is performed after defining requirements and contain information about system functionality. It is connected with usability testing, because information about system usability is always based on system functionality.

3. Usability testing validates system usability according to defined functional specification and defined specification of system's usability. First usability testing should be performed before developing user interface.
4. Internal design verification defines if the developed functional design is correct: were all the requirements fulfilled in this design or not.
5. Code verification: syntax and semantic analysis of programming code.
6. Usability testing is performed after code verification in order to find errors in implementing system's interface.
7. Unit testing checks how well separate modules work
8. Integration testing ensures that separate modules can work together in a proper way.
9. Usability testing is performed again to ensure that system interface was developed according to defined usability specification.
10. Function testing checks if system functionality was developed according to functional design specification or not.
11. System testing is the process of attempting to demonstrate that a program or system does not meet its original requirements and objectives, as stated in the requirements specification (Kit, 1996). System testing is one of the most difficult types of testing, because it is hard to ensure that all the requirements were met and objectives were reached. Acceptance testing usually is provided after system testing (SDT Model presents these levels at one phase) and the main purpose of this testing phase is to compare the end product to the current needs of the users.

SDT Dotted-U Model views testing as a process of validation and verification.

Testing validation activities within SDT Dotted-U model ensures that the product satisfies its requirements – that the product is right. Testing validation activities have brown color on the SDT Dotted-U Model (Figure 3.1.1).

Testing verification activities ensures that the product is developed right. Testing verification activities have green color on the SDT Dotted-U Model (Figure 3.1.1).

Software testing phases, chosen for study are testing validation activities: unit testing, integration testing, functional testing, system testing, acceptance testing and usability testing; because this thesis focuses on phases and methods which help to build right product.

### 3.2 Unit testing

Unit testing is testing of individual units or groups of related units (IEEE/ANSI,1990).

Unit testing is a type of low-level testing that requires the knowledge of an internal structure of a program and is usually performed by the developing team. The main purpose of Unit testing is to ensure that appropriate module works according to it's specification.

Kit mentioned that unit testing manages the combinations of testing: it facilitates error diagnosis and correction by development and it allows parallelism, in other words, testing multiply components simultaneously (Kit, 1996).

Typical way of managing test for single module is provided in the Figure 3.2.1

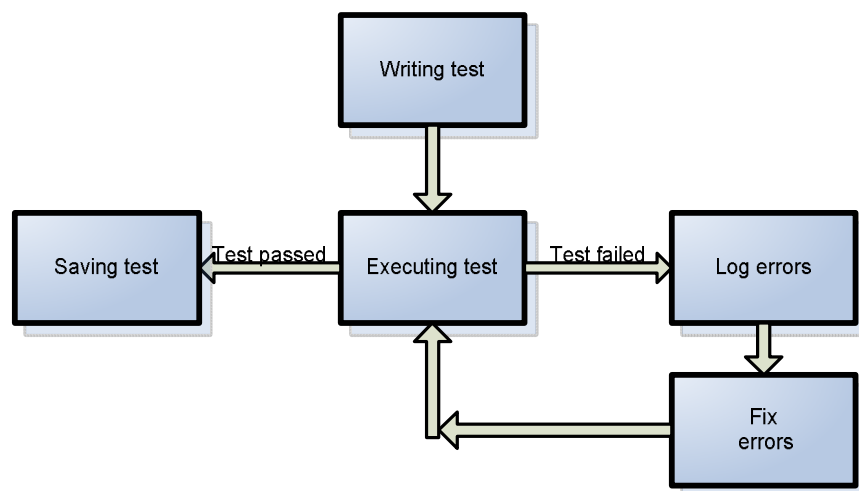


Figure 4.2.1 Managing single test.

First developer has to write test and then execute it. If test is passed, developer should save it and continue testing with new test, if test is failed, errors should be logged, then fixed and test should be executed again till the moment it will be passed. Logging errors and saving old tests are very valuable actions, because they will help in fixing errors in future. Ideally all test cases are different.

There are different Unit testing tools, provided for different programming languages. For example JUnit is a programming tool that provides unit testing facilities for programs, written on Java language.

The main benefit of unit testing is that it provides contract that single module should follow. As a result it helps to find errors in the development stage. The main disadvantage of unit testing is involving developer into the test process – developer has to spend his time doing testing activities and testers document test cases and found errors.

Unit testing can be applied to different software development models and especially in flexible methodologies such as XP and Scrum, where software can be developed using Test Driven Development (TDD) approach. TDD approach suggests developing first test and then code, this practice will help to ensure that code was implemented as it was required.



### **3.3 Integration testing**

Integration testing is testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them (IEEE/ANSI,1990).

Integration testing is second type of low-level testing, which is applied after unit testing and it validates the mutual work of separate modules, tested on unit testing phase. According to Kit, the primary objective of integration testing is to discover errors in the interfaces between the components (Kit, 1996). In general integration testing does not mean testing modules of a program, it means integration and testing together various components of some unit. That's why there are different levels of integration testing, such as testing programs of a subsystem, testing subsystems of a system, or the most often level testing modules of a program.

There are two types of integration testing – incremental and non-incremental.

#### **Incremental integration testing**

Incremental type supposes testing new module with already tested modules, after this module was tested, it is added to tested set. There are two approaches in incremental testing: bottom-up and top-down.

Bottom-up testing starts from the lowest module of the system and the main rule is that to be eligible to be the next module all of the module's subordinate modules must have been tested previously (Myers, 2004).

Top-down testing starts from the main module of the system and main rule is that to be eligible to be the next module, at least one of the module's superordinate(calling) modules must have been tested previously(Myers, 2004).

For illustrating top-down and bottom-up methods it will be convenient to use program, structure of which is provided on Figure 3.3.1.

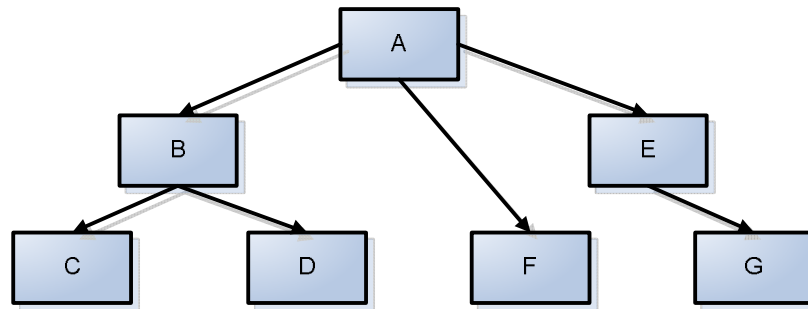


Figure 3.3.1. The structure of the program.

Figure 3.3.1 provides the basic structure of the program – it shows main modules of the program and how they are related. Arrow going from module A to module B means that module B is used in module A. The lowest module at this example are C,D,F,G, the highest – A. Bottom-up testing can be started from any of the modules at the lowest level, f.e. from C, top-down testing starts from the highest module A.

Both approaches have their own advantages and disadvantages. For example, top-down testing has the advantage, that skeletal version of the program can exist early and allows demonstrations(Kit, 1996), but the main disadvantage of this approach is that stub modules should be generated and these modules are often more complicated than they first appeared to be (Myers, 2004). Bottom-up approach is advantageous if the major flaws occur toward the bottom of the program, but the main disadvantage is that it can't operate with the whole program till the last module is added (while top-down operated with whole program from the very beginning, changing modules on stubs).

### **Non-incremental testing**

Non-incremental or 'big bang' integration offers testing each of the modules as a stand-alone program, then tested modules are combined. Testing of each module requires a special driver module and one or more stub modules (Myers, 2004). For example provided

in Figure 3.3.1, when testing module B one should generate test driver for this module, that will drive test cases through this module and stub modules for modules C and D. Comparing to incremental testing, “non-incremental testing” requires more work, because it requires more number of test drivers and stub modules to be created. The comparison of these approaches for example, provided on figure 3.3.1 is presented in table 3.3.1.

Table 3.3.1. The number of test drivers and stub modules for program, provided on fig. 3.3.1

	Non-incremental testing	Incremental testing	
		Top-down	Bottom-up
Number of test drivers	6	0	6
Number of stub modules	6	6	0

As integration testing is a second phase of low-testing after unit testing, it is usually applied together with unit testing and has the same disadvantages: for providing integration testing one should know the internal structure of the program, usually this type of testing is accomplished by developer, as well as unit testing.

### 3.4 Function testing

Function testing is testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions (IEEE/ANSI,1990). Function testing exists to ensure that all the implemented functionality acts according to the defined functional specification.

According to Kit, all black box methods for function based testing are applicable (Kit, 1996).

#### **Black-box methods**

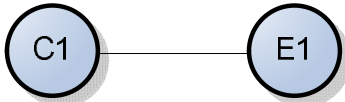
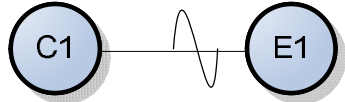
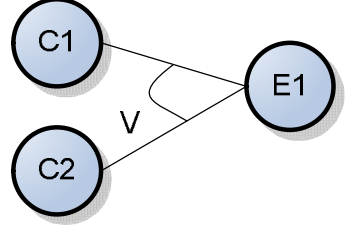
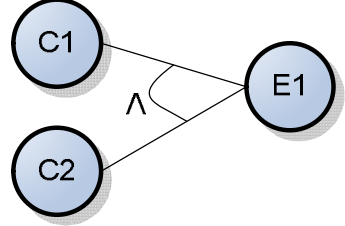
According to Roe, black box refers to testing which involves only observation of the output for certain input values; that is there is no attempt to analyze the code which produce the output (Roe, 1987). That means that test cases in black box testing are developed from functional specification of the program without taking into account internal structure of the program. The primary purpose of any method of black box testing is to find the maximum number of errors using the minimum number of test cases.

The following methods of black box testing are usually used:

- Random input testing – is the most simple type of black box testing, input data is generated randomly.
- Equivalence partitioning technique offers to develop test cases from input data, divided on partitions; test cases are supposed to cover each of the partitions at least one time.
- Boundary value analysis technique uses extreme input data values for test cases: minimum, maximum, error values and etc. Boundary analysis has similar logic with equivalence partition – it also uses partitions of input data, but these partitions are applied on “corner cases”.
- Cause effect graphing is a “systematic method of generating test cases representing combinations of conditions” (Omar, 1991). Cause-effect graph is a directed graph,

which maps a set of inputs to a set of outputs. Usually inputs are presented on the left side and outputs on right side.

Table 3.4.1 Basic parts of cause effect graph

Graphic notation	Meaning
	Identification
	NOT operation
	OR operation
	AND operation

According to Myer, cause effect graphing consists of the following steps:

1. input conditions and output effects are defined for programming module, that is tested;
  2. cause-effect graph is developed
  3. transforming cause-effect graph into a decision table
  4. converting decision table rules to test cases - each column of a decision table represents a test case (Myer, 2004).
- The condition table method is based on creating condition table, which columns present combinations of conditions that can occur in the program. These conditions (causes) primary appears from program specification, but usually they are also

added during testing process. Then program is executed and conditions are compared to the results (effects). The method concentrates on program specification, especially on what program should do, rather than what it should do not.

Assume table 3.4.2 describes conditions and results that appear during testing process.

Table 3.4.2 The condition table method. Input and Output conditions.

Causes(Input conditions)	Effects(Output conditions)
C1: enter One	E1: Message A is displayed
C2: enter Two	E2: Message B is displayed
C3: enter Three	E3: Message C is displayed

Table 3.4.3 Example of condition table

Causes section	C1	1	0	1	1
	C2	0	1	1	1
	C3	0	0	0	1
Effects section	E1	1	0	0	1
	E2	0	1	0	1
	E3	0	0	1	1

Table 3.4.3 provides example of condition table. Each column of condition table performs the test case to be evaluated. The input conditions for test cases are taken from Causes section and obtained outputs are compared with results in Effects section. For example, when evaluating first column, one have to enter One (as C1 has value 1) and the expected result is that message A is displayed (E1 has value 1). There are a number of algorithms for generating condition tables, usually these algorithms are based on binary graphs of tested program.

Function testing is closely connected to other types of testing, such as regression testing and usability testing. Regression testing checks how new functions or code modifications impact on existed program functionality. Function and regression testing types are applied during the whole cycle of program development.

The main document for performing function testing is functional test plan. Usually test plan is developed by lead tester of the project. Test plan defines the requirements of function testing, test strategy, assess risks, identify human recourses and schedule.

Test plan is used by testers for evaluating test strategy, the results obtained after test cases executing – functional test results are saved. These results are used by reviewer (project manager, lead tester) for making report to the development team and business community. Use case diagram, illustrating roles and responsibilities in functional testing, is provided on figure 3.4.1. The idea for diagram was taken from professional IT resource Developer.com (Developer.com, 2010).

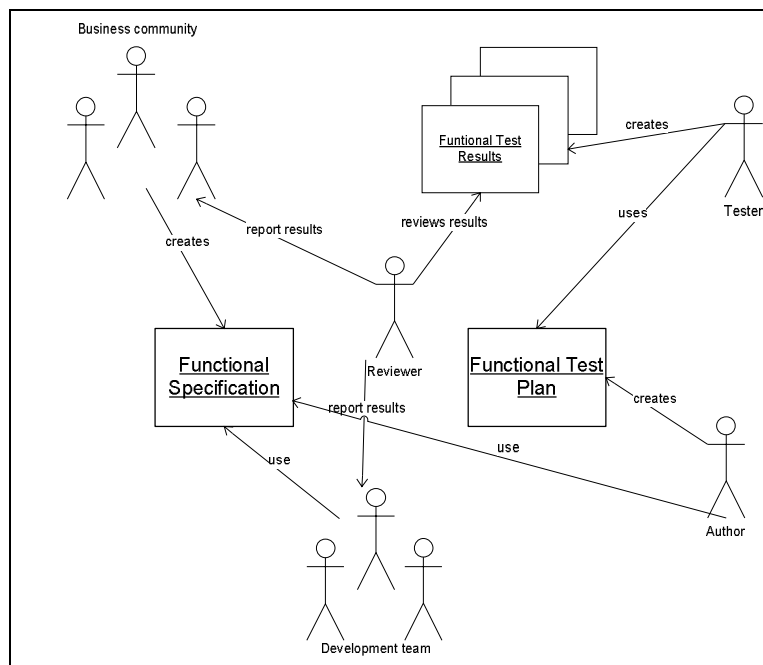


Figure 3.4.1 Roles and responsibilities in function testing

Functional testing is often performed with black-box methods, but these methods not always provide the appropriate validation for the program. Chapter 4.3 describes the method which can be used for more effective functional testing.



### 3.5 System testing

System testing is a testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements (IEEE/ANSI, 1990).

System testing is the most difficult type of testing process. System testing does not perform testing of program's functions (it is performed by functional testing), "system testing has a particular purpose: to compare system or program to its original objectives" (Myers, 2004, pg. 110). System testing is a destructive testing process that tests not only the design of the program and checks how system's specification is fulfilled, but also ensures that objectives of the system are met (even such immeasurable objective as customer satisfaction).

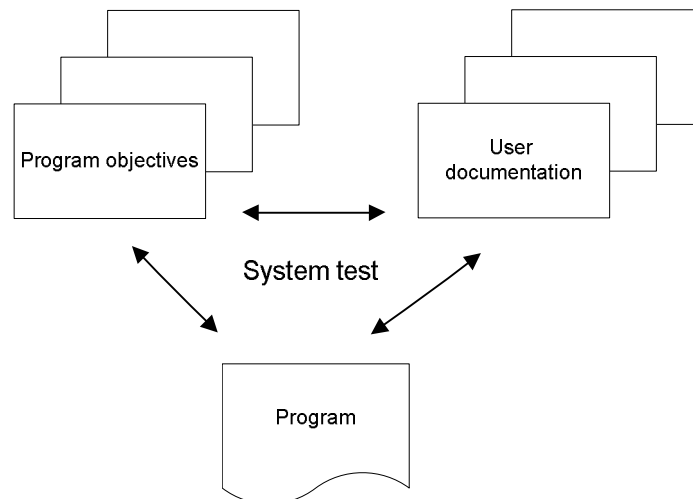


Figure 3.5.1 System testing (Myers, 2004).

Figure 3.5.1 shows that system testing is performed between defined objectives of the system, user documentation and system itself. If one of these components is absent, system testing cannot be performed. System testing is the most difficult type of testing process, because there is no test-case-design methodologies for comparing program objectives to the program. This comparison is hard to be performed, because objectives contain only information what and how should program do, but not specify the representation of program functions.

Different type of test cases can be used during system testing. Myers defined 15 categories of test cases that should be explored when designing test cases: facility testing, stress testing, usability testing, security testing, performance testing, storage testing, configuration testing, compatibility testing, installability testing, recovery testing, serviceability testing, documentation testing, procedure testing (Myers, 2004).

Facility testing checks if all the facilities, defined in program specification were implemented. Volume and stress testing check the ability of the program to work with heavy volumes of data and heavy load. Usability testing validates user interface, checks that it is suitable for the user. Security testing checks how security issues are implemented in program. Performance and compatibility testing types try to find cases when program does not satisfy its performance and compatibility objectives. Configuration testing is made with programs that can be configured (each possible configuration should be tested). Installability testing checks installation procedures of the system. Recovery testing checks how the system recovers from the errors. If the program has objectives to serviceability, the fulfilling of these objectives should be checked in serviceability testing. Documentation testing checks how accurate user documentation is. Procedure testing checks how prescribed human procedures can be performed in the program.

All these types of testing are used for system testing, but not all of them are used always. Which of the categories of system testing should be used is defined founding on the type of a program and its specification.

### 3.6 Acceptance testing

IEEE/ANSI Glossary provides two definitions for acceptance testing:

1. formal testing conducted to determine whether or not system satisfy its acceptance criteria and to enable the customer to determine whether or not to accept the system.
2. formal testing conducted to enable a user, customer, or other authorized entity to determine whether to accept a system or component.

Acceptance testing usually is performed by customer and the main purpose of it is to ensure that the product meets the needs of a user. Acceptance testing involves running and operating the software in production mode for a pre-specified period (Kit, 1996). Acceptance testing is connected with terms alpha and beta testing.

Alpha testing is a type of acceptance testing which is performed inside the development company with participating end users. Alpha testing is useful, because target users has possibility to try the real product and give their feedback.

Beta testing is a type of acceptance testing which is performed outside the development company with participating defined subset of target users. Beta testing is usually provided before making product available for all target users. Beta testing is more effective than alpha, because the product can be tested in “real world”, the lacks, which cannot be found during alpha testing, can be identified during beta testing. Also beta testing allows involving staff in education of developed software.

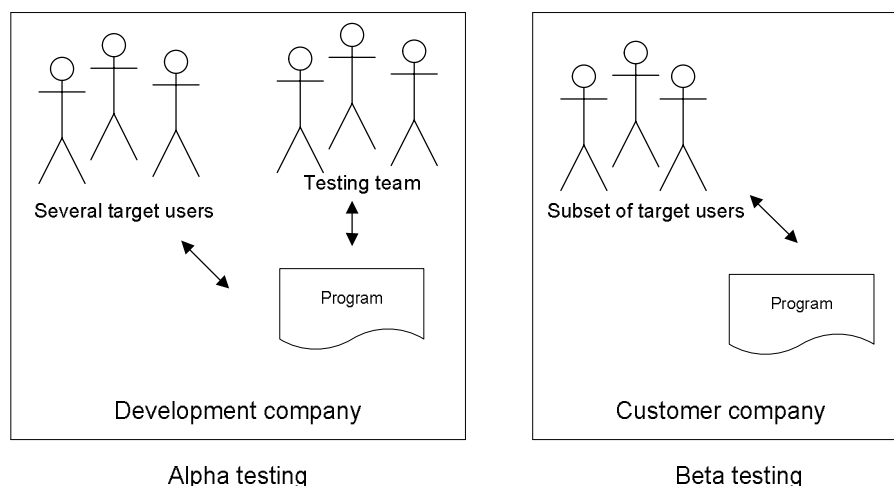


Figure 3.6.1 Alpha and Beta testing.

Figure 3.6.1 illustrates alpha and beta testing. Alpha testing allows development organization to perform testing with participating testers and several target users of the system. As alpha testing is performed in development organization, it is easier to fix errors if any occurs. Beta testing is performed without participating testers of Development Company and is accomplished on the customer's side. It allows involving more target users and working with software on practice, but it would take more time to eliminate the lacks that were found out during beta testing.

Based on validating how software satisfy customer's requirements, acceptance testing focuses on verifying man-machine interactions, required function features, and specified system constraints (Hsia et al, 1994).

### **3.7 Usability testing**

Usability is the ease with which the user can operate, prepare inputs for, and interpret the output of the system and components (IEEE/ANSI,1990). Usability testing focuses on measuring usability of the software involving target users in the testing process. In other words it validates how suitable is software for the user. High level of usability can be reached applying user-centric design (UCD) approaches.

According to ISO 13407 standard (ISO, 1999), which provides guidance on achieving quality in use, the process of developing usability model consist of the following steps:

1. Planning the human centered process: the plan of the human centered process is provided, methods for defining user expectations are determined.
2. Specifying the context of use: the context of use methodology is specified. Different methods that can be applied are: interviews, observations, brainstorming, scenarios and etc.
3. Specifying user and organizational requirements: requirements can be specified with methods mentioned in previous step: interviews, observations, scenarios and etc.
4. producing design solutions: designed solutions are implemented,
5. evaluating design solutions against user requirements

User evaluation can be performed by applying different evaluation sets: heuristic evaluation lists, which contain evaluations concerning navigation, application logic, context awareness and others; Nielsen heuristic lists and etc.

Usability model evolution process, provided in Figure 3.7.1 is iterative: if user requirements are not satisfied, the process continues.

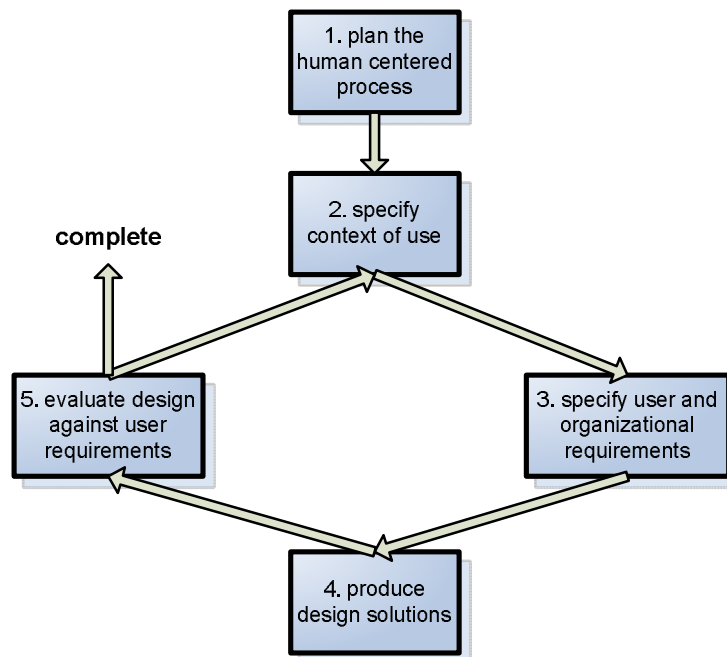


Figure 3.7.1 The interdependence of user centered design activities (ISO, 1999).

Usability testing is experimental method, based on the interviewing target users (participants) of the system according to the defined scenario. Moderator is the person who is responsible for performing usability testing and saving results.

Usability testing contains following steps:

1. Participants for interviewing are elicited with special questionnaire that helps to recognize target users of the system.
2. During interview moderator asks participants to accomplish different tasks concerned to tested software.

The main point of usability testing is defining complex situations which user meets during program exploitation. The main complexity in usability testing process is big amount of information flows, which should be recorded: face mimics of participant, screen projection with user's actions, user's reactions and etc. These flows should be then synchronized to get the whole picture. Obtained results are used to make program interface more obvious for user.

## **4. SUGGESTION FOR A REPERTOIRE OF TESTING METHODS**

At present there are many software testing methods that can be applied at different testing phases for estimating the quality of the developed software. Applying concrete methods in many respects depends on the type of software.

The initial objectives for this thesis were to provide a sufficient literature review on different testing phases and for each of the phases define the method that can be effectively used for improving software's quality. This section provides information about software testing methods that can be applied on the phases defined in section 3.

The section presents information in the following manner: the problem, concerned to each of the phases is identified; the method that can help in eliminating this problem is suggested and particularly described. The suggestion of the method is based on the literature review. The order of phases is similar with the section 3.

### **4.1 Unit Testing**

Unit Testing is cost effective and valuable for any system, because it helps to find serious mistakes, which can be hardly found using other types of testing. Unit Testing is a rather complex type of testing and the main difficulty is to define appropriate test cases for a module.

Test cases are usually developed basing on the internal structure of the module. The problem is that it is difficult to build all necessary test cases directly from the programming code, because it is very simple to miss the branch that can occur in the program, which can contain errors. Besides ideally each of these test cases should be unique. This condition is hard to accomplish if one have to deal with a textual view of the module, because it is hard to estimate routes which test cases should consider. The decision of this problem is using graphical representation of the module, i.e. programming code should be converted to the graph.

One of the methods, which help to make graphical representation of programming code for developing test cases was provided by Zang (Zang et al., 2009). The main idea of this method is to make a representation of textual view using Coloured Petri Nets. This approach provides following benefits: it allows to get graphical structure of the programming code and allows to perform test case firstly on the paper. Comparing to other graphical representations of program's textual view, such as more simple representation based on the binary tree, this method allows to build dynamic graphical representation, as it manipulates developed representation as a Petri Net.

This section provides definitions of Petri Nets and Coloured Petri Nets, that is necessary for describing suggested method. After definitions, the method for transforming textual view to graphical using Petri Nets is provided.

### **Petri Nets**

Petri Nets basically is a mathematical modeling language that helps to describe distributed systems. Petri nets does not change the net structure, the concept of Petri net transformations is a rule-based approach for dynamic changes of the net structure of Petri nets (Cordic, 2008).

**Cardoso** mention that Petri Net can be viewed from three aspects:

- As a graph with two types of nodes (the places and the transitions) and a token game defining the evolution;
- As a collection of vectors whose components are natural numbers and whose behavior can be characterized by linear programming;
- As a production rule system based on specific rules of form (Cardoso, 1999).

Graphically places are presented as circles and transitions as bars; directed arcs describe which places are preconditions of post conditions for appropriate transition. Places may



contain natural number of tokens, distribution of these tokens in Petri net is called marking. Graphical presentation of subnet is doubled circle.

The example of Petri Net is provided on Figure 4.1.1

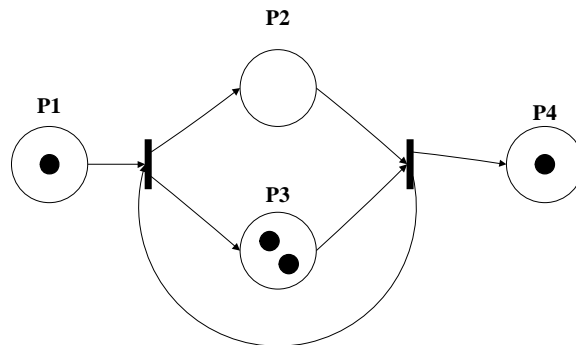


Figure 4.1.1 Example of Petri Net

Coloured Petri Net (CP-net) is a graphical oriented language for designing systems, which joins advantages of Petri Nets (provides primitives for process iteration) and high-level programming languages (provides primitives for definition data types and manipulation with data values). The extension of Petri Nets to Coloured Petri Nets is in providing additional information to the elements of the net:

1. Tokens are transformed to the objects which may contain one or more parameters.
2. Places are supplemented with information about types of tokens that can be located in this place.
3. Arcs, which come from places and transitions, are supplemented with information about token types that can participate in transition initiation.
4. Information about variable's value is added to the initial marking of the net.

### A method of transforming code to the graphic view using Petri Nets

This method was presented by Zang et al. in the paper “Unit Testing: Static Analysis and Dynamic Analysis” (Zang et al., 2009).

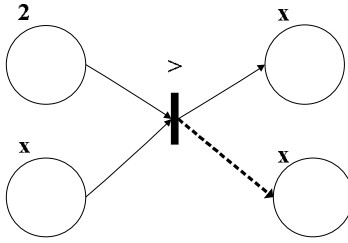
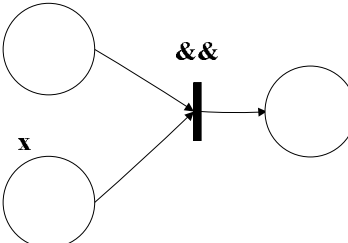
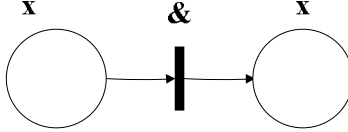
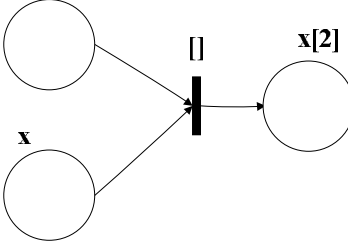
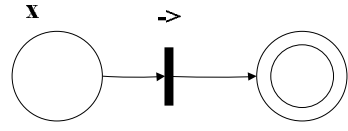
In this approach Unit Testing activities are divided into two parts: static analysis and dynamic analysis. Static analysis makes validation of syntax and semantic of the code, while dynamic analysis performs black-box and white-box testing.

The first step of the method is transforming code of the program to a Coloured Petri Net. For doing this it is necessary to define transforming rules for different parts of the program, such as variables, condition operators, cycle operators and etc.

Several types of operators are defined; each transition in Coloured Petri Net must contain only one of these operators. The table that consist information about defined operators is provided below.

Table 4.1.1 Coloured Petri Net for operators

Operator type	Description	Coloured Petri Net Example	Comment
Assign Operator	=		The value in pre-place is passed to the variable in post-place.
Operation Operator	+, -, *, /		Two values from pre-places are calculated and moved to the variable in post-place.

<p>Compare Operator</p>	<p>&lt;, &lt;=, &gt;, &gt;=, ==, !=</p>		<p>Values from pre-places are compared, first arc-out denotes true result and second(dotted) - false.</p>
<p>Logic Operator</p>	<p> , &amp;&amp;</p>		<p>Transition calculates logical operation from two pre-places and sends the result to post-place.</p>
<p>Address Operator</p>	<p>&amp;</p>		<p>The address from pre-place is assigned to post-place.</p>
<p>Array Operator</p>	<p>[]</p>		<p>Transition define the array element in post-place.</p>
<p>Trigger Operator</p>	<p>-&gt;</p>		<p>Only signal is passed to the post-place to continue the execution.</p>

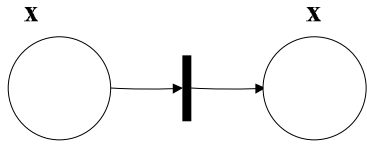
Empty Operator			The value of the variable from pre-place is send to the variable of post-place.
----------------	--	--	---

Table 4.1.1 provides information about basic operator's representation in Coloured Petri Net. This logic of this representation is used for defining representation for more complex operators.

For example, graph for switch operator can be defined:

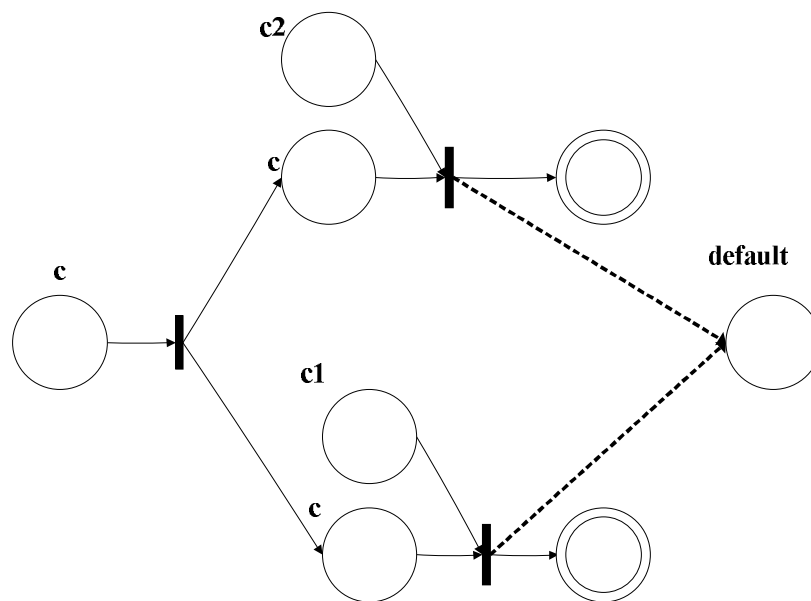


Figure 4.1.2 Switch operator

Here 'c' is control variable, which is compared with different cases – 'c1' and 'c2' default operation is called when control variables was not matched to any of the cases.

So one can build a Coloured Petri Net for a module.

Zang provide theorem for developing test cases. The theorem asserts that each test case must be one of the following graphs: non-closed graph, retreated closed graph, non-retreated closed graph (the proof is omitted) (Zang et al., 2009).

Closed graph is a graph that can be walked from root vertex to other vertexes once and back to the root vertex. Graph is called retreated if there are two directed edges associated with same starting vertex and there are two directed edges ending with the same ending vertex; a closed graph is called non-retreated if there is a vertex that is both starting and ending vertex.

The main benefits of the method, provided above are:

- Method allows to make a graphic representation of a module;
- Graphic presentation obtained with Petri Nets provides primitives for process operations unlike binary graphs;
- Method provides both static and dynamic analysis of the unit.

This method can be applied for another purposes of transferring code view to graphical, but it is can be especially successfully applied in unit testing, where the programming module usually has not very big size (number of LOC) so it is easier to get graphical structure of module. Using Petri Nets allow managing the process of program execution.

## 4.2 Integration testing

Integration testing is the second phase of low-testing. There are three main approaches for providing integration testing: incremental top-down approach, incremental bottom-up approach, and non-incremental approach. These approaches were discussed in section 3.3.

The problem, identified for integration testing phase is concerned with object-oriented programming. At present time object-oriented programming becomes more and more popular. The reason of it is that “object oriented programming takes the best ideas of structured programming and combine them with several new concepts” (Shildt, 2003) and basically object-oriented paradigms provides more realistic representation of existed entities to the code entities. Object-oriented programming is supported by many programming languages; the most popular are C++, C#, and Java, as well as many different tools that can be used for analyzing and testing OO programs.

The problem of integration testing in object-oriented programming is that traditional methods are not appropriate for object-oriented programming. The reason is that object that is the module in object-oriented programming is more than a programming module in common sense. Usually it has state (attributes) and behavior (methods). Objects can communicate with other objects by sending messages. Traditional integration testing methods doesn't take into account the structure of OO programs; performing integration testing using these methods is difficult and inefficient.

In this chapter one of the methods of integration testing for OO programs is discussed. This method was developed by Zhe Li and Tom Maibaum and it allows generating test cases for integration testing from UML diagrams. The test cases for this method are assumed to consist of three parts: testing that the sequence of message calls conforms to the relevant sequence diagram; testing parameters; and testing object interactions by examining the states of objects after execution of prescribed sequences (Li, 2007).

Test cases in the suggested approach are implemented using the concept of coordination contract. The idea of coordination contract was provided by Andrade (Andrade, 1999). The main idea of coordination contract is providing information about objects within the contract and defining the rules of object's coordination. Coordination contract contains parts of the programming code and can be easily transferred to the program.

Next section provides information about coordination contract in more details, after that method for performing integration testing for OO programs is particularly described.

### **Coordination contract**

Contract is proposed as an extension of association classes at the representation level, which relies on implementation mechanisms that ensure the degree of flexibility required by the need to reflect changes in the business rules (Andrade, 1999). This flexibility is provided by using the mechanism of superposition instead of using mediators for coordinating interactions between objects.

The example of coordination contract, written on OBLOG language (UML-compatible language) is provided below, Figure 4.2.1.

```
contract ContractExample
participants
    p1: Participant1;
    p2: Participant2;
attributes
    double attr;
coordination
ExampleRule:
    when*->>p1.invokeMeth(n)
    with (p1.getState() + attr > n)
    failure {
```

```
//Java guard failure actions;  
//through an exception;  
};  
do p1.invokeMeth2()  
end contract
```

Figure 4.2.1 Contract example (Li, 2007).

The code above consists of two parts: components or participants – p1 and p2; and contracts - ContractExample. Coordination section contain one rule – ExampleRule, which has a trigger (the code after **when** keyword), optional guard (after **with**) and optional body (after **do**). The code in body section is executed only when trigger event happens and guard returns **true**. If the guard is **false**, failure is executed.

#### **Method for performing integration testing using UML diagrams**

This section describes how test cases for OO programs can be generated with UML diagrams. The main purpose of test generation is detecting failures connected with interactions between objects. The approach, provided by Li(Li, 2007) allows generating test cases using UML class and sequence diagrams. As has been already mentioned, test case generation contains three parts: testing sequences of message calls, testing parameters and testing object interactions.

Testing sequences of message calls validates if the order of methods, which were called is right. For doing this UML sequence diagram is used.



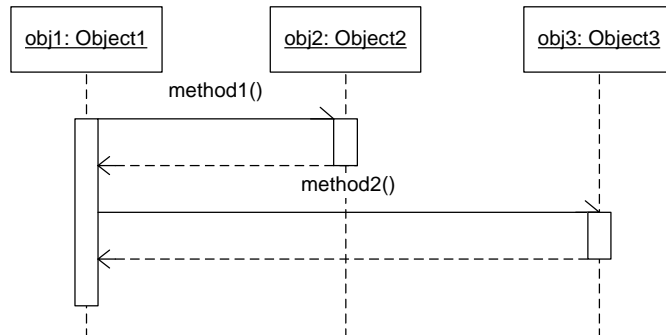


Figure 4.2.2. Example of UML sequence diagram

The idea of testing the sequence of invoked methods is quite simple. For doing this one should define the variable for each of the methods, participating in interaction. For instance its name can be step. This variable has several values; each of these values is associated with one of the methods. Firstly the variable step is initialized with the value of the method, which has to be invoked first. After that next the control goes to another method, but before invoking, method checks the value of step from previous method. Each of the methods “knows” the value of the step for the method, which has to be invoked before itself. If these values aren’t matched, the sequence is wrong; otherwise the control goes to another method.

Testing parameters activity is used for ensuring that parameters, appearing in messages are consistent within the diagram (Li, 2007). The algorithm of testing parameters for first message is as follows:

- saving parameters for the first message one by one
- get the type of each parameter obtained on the 1st step from class diagram(one should find the appropriate message in class diagram and define the type of the parameters)

- go through other messages from sequential diagram sequentially, if the parameter with the same name is found, compare the type of this parameter with the saved one. If types are mismatched, test fails, else, test passes.

Testing object interaction uses the simulation technique to check the post conditions of each object after all the interactions defined in a sequence diagram are completed (Li, 2007).

The main idea of testing object interaction is simulating the execution of the program of the sequence diagram. For representing this simulation, firstly the copies of all the objects, participating in the sequence diagram are created. After that on the other hand the expected sequence of sending messages and changing object state is accomplished and on the other hand the program is executed as it was designed.

The last step is to compare the states of the objects from the first test (when simulating the program execution) with second test (when the program was executed).

### **Test Case Implementation using contracts**

Test cases discussed above deal with interactions between objects. As the rules in the contracts can superpose the behavior of the components without changing their implementations, one can write test cases with coordination contracts.

The example of developing test cases with coordination contract is provided below.

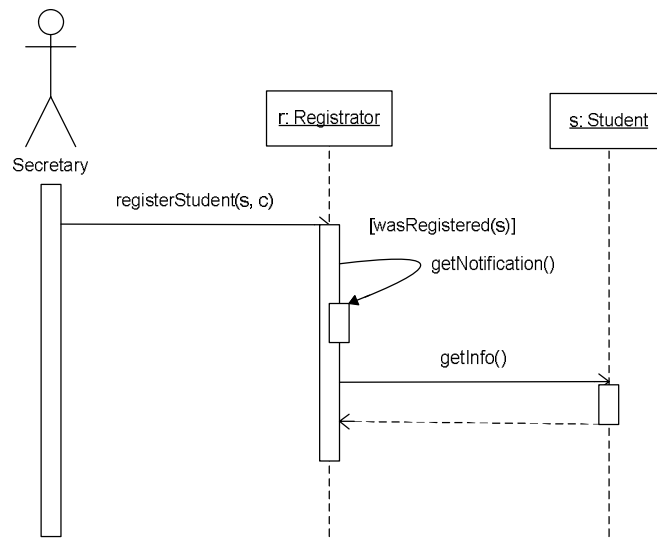


Figure 4.2.3. Example for developing test cases with coordination contract, sequence diagram

The basic system of a student course registration was used as an example for illustrating how to develop test cases using coordination contract. The actor Secretary invokes the method `registerStudent(s, c)` to register student `s` to the course `c`. The object `r` (which has type `Registrator`) checks if student has been already registered or not, calling method `wasRegistered(s)`. If student has been already registered to the course, `r` invoke method `getNotification()` and return 0, that means that `s` was registered, otherwise `r` invoke method `getInfo()` of `s` object to get info about student. After info is received, `r` registers `s` to the course `c`. The programming code for developing test cases using coordination contract is provided below.

### Contract for testing sequences of message calls

```
contract registration_TSMC_test
participants
    r: Registrar;
    s: Student;
attributes
    int result = 0;
    int step = 0;
coordination
CheckStep1:
    when*->>r.registerStudent(s1,c) && (s == s1)
    before{step = 1}
CheckStep2:
    when*->>!r.wasRegistered(s1) && (s == s1) && (step == 1)
    failure{
        r.getNotification();
        result = 0;
    }
    do{
        before{step = 2}
    }
CheckStep3:
    when*->>s.getInfo() && (step == 2)
    before{step = 3}
    after{
        if(step == 3){
            result = 1;
            step = 0;
        }
    }
end contract
```

Figure 4.2.4. Example of TSMC using coordination contract

The example provided by Figure 4.2.4 shows how to check the sequence of message calls. Each time when the message call occurs, the value of variable `step` is changed. When the next message call occurs, contract checks the value of the variable – if its value equals the code of current message call, the sequence is right, otherwise – wrong.

### **Contract for testing parameters**

```
contract registration_param_test
participants
    r: Registrator;
    s: Student;
attributes
    int result = 0;
    Student expectedStudent;
    Boolean precondition = false;
coordination
Precondition:
    when*->>r.registerStudent(s1,c) && (s == s1)
    before{
        precondition = true;
        expectedStudent = s1;
    }
Check_Parameter1:
    when*->>!r.wasRegistered(s1) && (s == s1)
    do{
        before{
            if( expectedStudent != s1 && (precondition)) {
                result = 0;
            }
        }
    }
}
```

```

Check_Parameter2:
  when*->>s.getInfo()
  before{
    if( expectedStudent != s1 && (precondition)) {
      result = 0;
    }
  }
  after{
    if( expectedStudent == s1) {
      result = 1;
    }
  }
end contract

```

Figure 4.2.5. Example of testing parameters using coordination contract

Figure 4.2.5 provides example how to use coordination contract for testing parameters. Given example stores information about parameter that was used in first method (s1) and each time when this parameter is used checks if it is the same parameter or not. If the parameter is the same, test passes, otherwise – fails.

### **Contract for testing returned value**

```

contract registration_param_test
participants
  r: Registrator;
  s: Student;
attributes
  Student s_copy;
  Boolean isEqual;
coordination
Test_Return_Values:
  when*->>r.registerStudent(s,c)

```

```

before{
    s_copy = new Student(s);
}
after{
    if( !wasRegistered()) {
        s_copy.getInfo();
    }
    if( s_copy.getCourses() == s.getCourses){
        isEqual = true;
    }
}
end contract

```

Figure 4.2.6. Example of testing returned value using coordination contract

The contract for testing returned value is based on method testing object interaction that was defined in section “Test code generation”. Contract creates copy of the object *s* and evaluates the sequence of message calls for this copy. After that two obtained objects are compared. If the objects are identical test passes, otherwise fails. For the given example it is hard to use this method, because if one will try to register the same student to the course second time, the result will be another (program won’t register student, it will send notification that this student has been already registered). So for testing the interaction for current situation, one shouldn’t save information about registered user in database, in this case test will be passed correctly.

The method of testing for OO programs with using coordination contract allows testers to provide integration testing of a program with UML diagrams. The method provides integration testing of OO programs from three points of view: testing the sequence of message calls, testing method parameters and testing the interaction between objects, all three types of testing are provided with coordination contract. The executable test cases can be generated from coordination contract using different tools for coordination contracts, for example Coordination Development Environment – tool for developing Java application using coordination contract.

### **4.3 Functional testing.**

Functional testing is often based on black-box method testing. The most applicable methods of black-box testing are random input testing, equivalence partitioning, boundary value analysis, cause effect graphing and the condition table method (Kit, 1996). The brief descriptions of these methods were provided in section 3.4.

The lack of traditional methods is that during the testing interior structure of the program is ignored and the only function that is considered while developing test cases is function evaluated by the program. Sometimes in program occur situations, when test case is passed successfully, but some variables or objects were set to wrong states, because some functions were incorrect.

Howdon proposed the method, which requires developing the complete set of functional tests for each of the functions participating in program design – “design functions”. This method requires accomplishing three main steps: identification of functionally important classes of input and output data; functional decomposition of data structures into design substructures; functional decomposition of programs into design functions(Howdon, 1980).

First step requires that all allowable values for the variables are indentified formally. Second step is defining design substructures, which are subsets of declared data structures and have a conceptual meaningful functional identity. Third step requires defining design functions and is the most complicated, because it requires deep understanding of program functionality.

Howdon divides design functions on three types. First type of design functions is functions which are used in typical program for forming functional capabilities of this program. The example is joining several functions for forming single program. Dividing programs on different modules of programming code allows to support program easily and reuse



program components if it is necessary. These modules are separated, evaluate concrete program's function and doesn't depend on other program functions. This type of design functions, Howdon called "parallel functional capabilities". Figure 4.3.1 illustrates the program with this kind of functions.

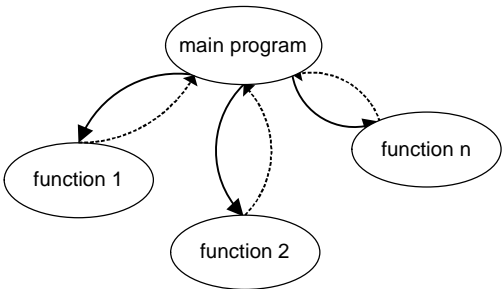


Figure 4.3.1. Parallel functional capabilities (Howdon, 1980)

Second type of design functions Howdon called "sequential decompositions into subfunctions". Programs are often divided into set of subfunctions that should be invoked sequentially. These subfunctions are also design functions according to Howdon and should be tested during the functional testing. Figure 4.3.2 provides the basic structure of the program with this kind of design functions.

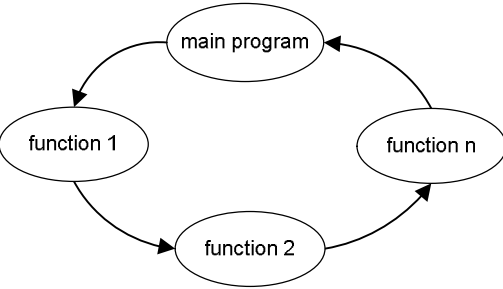


Figure 4.3.2. Sequential decompositions into subfunctions (Howdon, 1980).

In the example provided on Figure 4.3.2. main program invokes function 1, which requires invoking function 2, which invokes function n. In this example function n returns the result to main program, but in general it is not necessary.

First two types of design functions defined by Howdon are easy to recognize, because these functions “correspond directly to relatively independent pieces of code”. These functions are computational, as they provide capabilities for computing values.

Control functions are functions that are used for selecting the type of computation, terminating iterative or recursive process and evaluating other control functions.

Figure 4.3.3 provides an example of control function.

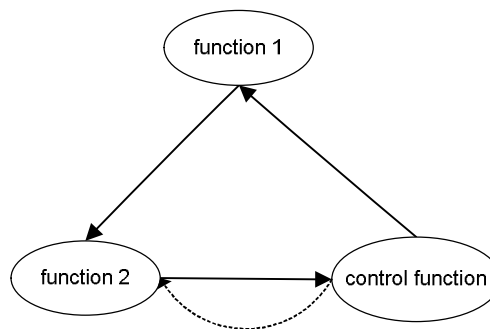


Figure 4.3.3. Control function(Howdon, 1980).

The example provided on Figure 4.3.3 illustrates basic case of using control function. Assume that there's function 1 in the program which invokes function 2. Function 2 has a loop which has control function inside of it, which will manage the control – either it will give the control to function 2 to continue loop calculations, or it will terminate the loop and return value to function 1. Howdon defines control functions as third type of design function, which also should be tested during functional testing.

### Testing functions in context

The example below provides testing design function f. Assume the symbols for the examples are:

P – program;

f – design function of P;

x – input variable of P and f,  $x \in (-\infty; \infty)$

Suppose that f in P is executed only when  $x > 1$ . If f is tested independently of its context in P, the extreme values for x will be:  $x = k$  and  $x = -k$ , where k is large. If f is tested within context, the condition that  $x > 1$  will be taken into account and the extreme values of x will be:  $x = 1$ ,  $x = k$ .

Howdon mention, that some classes of errors will not be discovered unless the context is taken into account during the generation of test data for the function. The functional context for the design function is defined by symbolically evaluated systems of branches predicates that appears along the program paths which lead to this design function (Howdon, 1980). Predicates describe the set of the input domain over which design function is used. If more than one path exists, it won't be possible to take the complete functional context.

According to Howdon, the importance of testing design functions is stressed because of several reasons:

- testing at program level doesn't allow to validate all important computational substructures of the program; many variables and data structures inside the programs are not functionally meaningful when program is viewed as a whole, in spite of they are important inside the design functions.
- design function testing can help to find errors, which can be skipped by black-testing methods, f.e. program with incorrect control function can work in a proper way, but not very efficiently.

The main disadvantage of the approach offered by Howdon is that the person, who performs this approach, has to have deep understanding of program's structure to be able to define design functions correctly.

#### 4.4 System testing

As it has been already described in chapter 3.5 system testing is the most complex type of testing and it involves several categories of testing types (f.e. Myers defined 15 categories). The complexity of system testing also proves the fact that there can be no design methodologies for test cases because requirements and objectives do not, and should not, describe the program's functions in precise terms (Kit, 1996).

There are different methods that can be used during system testing. In many respects these methods depends on software type, but there are also methodologies that can be used for big range of software systems and provide effective results. In this section, the method that performs system testing based on UML is suggested. The method uses sequence and statechart diagrams for generating test cases. One of the main challenges in system testing is coverage of the system states, because the number of system states is usually very big and system state model usually is not constructed. Method, proposed by Sarba and Mall allows to cover different system states using developed UML diagrams.

Sarba and Mall propose the method for designing test cases to achieve the coverage of the system states based on UML diagrams. This method requires getting the set of test cases for reachable states of the system and is quite suitable because it is based on developed during analysis phase UML diagrams.

##### **Methodology for designing test cases**

The pseudo code of proposed methodology is provided on Figure 4.4.1.

Algorithm Input: <i>SC</i> : The set of all scenarios of the system under test. <i>SD</i> : Sequence diagrams for each scenario in <i>SC</i> . <i>SCD</i> : State chart diagrams for all objects in the system. Output:
--

```

A test set  $tSet$  consisting of specific sequences of scenarios.
Begin
  //Initialization
1 count = 0, ns = 1,  $tSet = \Phi$ , sysState = NULL,
  isNewState = FALSE
  //Generate all possible sequences of ns scenarios from SC
  with repetition//
2 While (ns  $\leq$  MAX_SCN) Do
    //ns denotes number of scenarios in a sequence//
3   seqScenariosSet = GenerateSequence(SC, ns)
4   Initialize the objectStateList.
    //objectStateList is initialized with the initial state of all
    objects which can be known from SCD//
5   For each sequence  $S_i$ ,  $S_i \in seqScenariosSet$  Do
6     For each scenario  $sc_j$ ,  $sc_j \in S_i$  Do
        //Select a scenario in the sequence
7       For each message  $m_k$ ,  $m_k \in sc_j$  Do
            //Determine the state after execution of the
            message  $m_k$ //
            //Let message  $m_k$  be sent to object  $O_1$ 
8           Get the current state of  $O_1$  from objectStateList
9           Determine the next state of  $O_1$  that may be caused by the
            message  $m_k$  by examining SCD for the object  $O_1$ 
10          Update the state of  $O_1$  in objectStateList
11          If SearchSyState(objectStateList, sysState) == FALSE
            //SearchSyState(...) returns false
            //if objectStateList is not already covered
12            If  $S_i \notin tSet$ 
13               $tSet = tSet \cup S_i$  //Add this to test set
14               $sysState = sysState \cup objectStateList$ 
15              //The new state is covered and added to
              sysState//
24            count = 0

```

```

//Reset count since a new state has been
found//
17      isNewState = TRUE
18      EndFor
19      EndFor
20 EndFor
21 If (isNewState == FALSE) count = count + 1
22 If (count > δ) exit() //Terminate test case generation
23 ns = ns + 1
24 EndWhile
End

```

Figure 4.4.1 Pseudo code of proposed methodology (Sarba, 2007).

In the proposed methodology test cases are generated through the following steps:

1. All the use cases and scenarios associated with these use cases are defined.
2. Different sequences of scenarios are tested, reached system states are defined.
3. Based on step 2 a set of scenario sequences for covering all defined system states is chosen (Scenarios are chosen in an incremental manner – if one scenario doesn’t cover defined state, two scenarios are tested and ect.).
4. For each of the scenario sequences system state is defined (if it was not defined earlier). For identifying does scenario execution leads to new uncovered system state, methodology propose to use messages exchanged by participating objects.
5. Each scenario is selected as test case. This test case should lead only to states not reachable by other test cases.

**Practical usage of methodology**

This section provides the example of applying the methodology proposed by Sarba et al. Example is abstract, it doesn’t concern to concrete system, the main purpose of this example is to provide practical manual how to use defined methodology.

Assume that system has a set of use cases  $U = \{U_1, U_2, \dots, U_n\}$ .

Assume that a sequence diagram of use case  $U_i$  has a view provided on Figure 4.4.2.

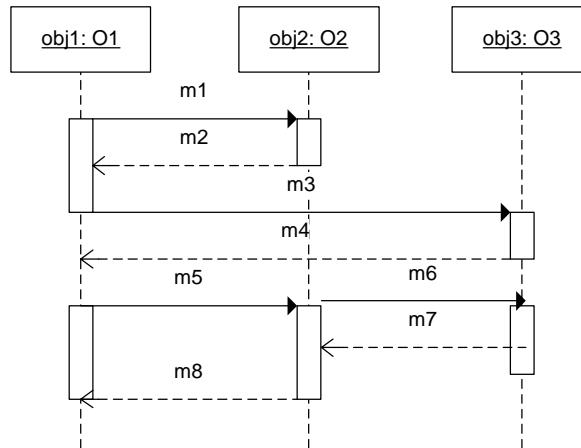


Figure 4.4.2 Sequence Diagram for use case U<sub>i</sub>.

O1, O2, O3 – objects that takes part in program

m1-m8 – messages, exchanged by objects

The sequence of steps defined in section “Methodology for designing test cases” is used in this example.

1. All the use cases and scenarios associated with these use cases are defined.

As we have the set of use cases we have to define scenarios associated with these use cases. Scenarios are defined from sequence diagrams. Scenario contains the sequence of interactions: each interaction contain the following information

<fromObj, toObj, message> ,

where

fromObj is name of the object that send message

toObj is name of the object to which message was send

message – name of the message

F.e. sequence diagram, provided on figure 4.4.2. has two scenarios:

<S1: <O1, O2, m1> <O2, O1, m2> <O1, O3, m3><O3, O1, m4>>

<S2: <O1, O2, m5> <O2, O3, m6> <O3, O2, m7><O2, O1, m8>>

2. Different sequences of scenarios are tested

All possible sequences of scenarios (SS), defined on the previous step should be tested:

SS1: S1

SS2: S2

SS3: S1, S2

3. Reached system states are defined.

As was mentioned earlier, each scenario consist of interactions, scenario is executed only when each of interactions, included in this scenario, is completed. When the interaction is completed, the object, participated in this interaction may change its state. All object's reachable states are performed by statechart diagram of this object. Next state of the object is determined from this diagram. Consider statechart diagrams, provided on Figure 4.4.3 are diagrams for objects O1, O2 and O3 participating in sequence diagram provided on Figure 4.4.2.

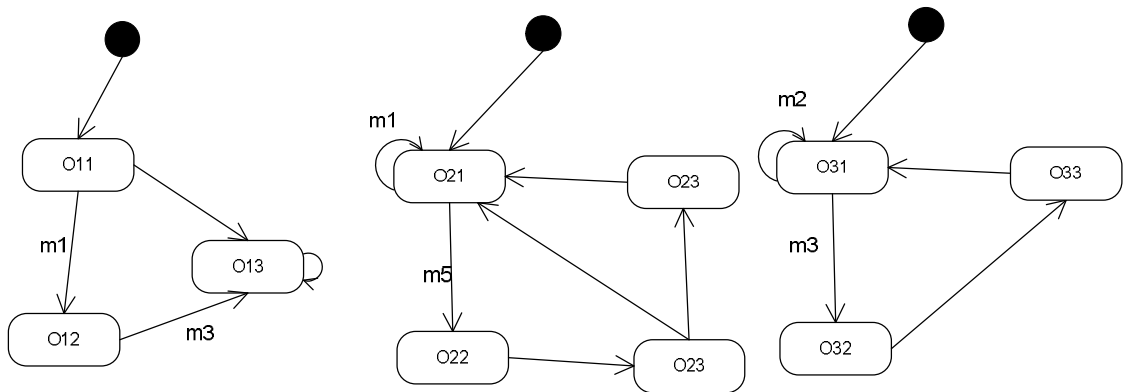


Figure 4.4.3. Statechart diagrams for objects O1, O2, O3

Diagrams provided on Figure 4.4.3 contain information about object's states. If there's an arc from one state to another that means that object can change its state from one to another, when the condition mentioned above the arc is accomplished. If arc doesn't have condition that means that this change of state is reachable, but not defined for this scenario. Table 4.4.1 provides information about changing states of objects for scenario S1.



Table 4.4.1 Object-Interaction table for objects O1, O2, O3

Object Interaction	O1	O2	O3
Initial state	O11	O21	O31
<O1, O2, m1>	O12	O21	O31
<O2, O1, m2>	O12	O21	O31
<O1, O3, m3>	O13	O21	O32
<O3, O1, m4>	O13	O21	O32

Last row of Table 4.4.1 contain information about state of objects O1, O2, O3 after executing scenario S1. These states will be initial for scenario S2.

#### 4. Defining test case suite

When scenario is defined and all the interactions within this scenario are known, one should go through the these interactions and check if any of them lead to the new state of the system that still haven't been covered by existed test cases. If not, one should create test case according to this scenario. One scenario can cover more than one system state.

Coverage of the system states is always a bottleneck in system testing, because the number of system states can be enormous and system state model usually is not constructed. Method, proposed by Sarba and Mall allows to cover different system states using developed UML diagrams. Using artifacts that have been already developed makes the usage of this method suitable. The main disadvantage of this method is that it can't be applied until required artifacts exist.

#### **4.5 Acceptance testing**

According to Kit, acceptance testing is the process of comparing the end product to the current needs of its users (Kit, 1996). In that way, acceptance testing is the last phase of testing which checks how well user requirements are satisfied. It is also one of the challenging and difficult types of testing, and in many respects it depends on the type of software. Different approaches can be applied for performing acceptance testing.

Yu mention acceptance testing relies on system requirements and the completeness of these requirements should be checked (Yu et al, 1999). For doing this it is suggested to perform acceptance testing basing not only requirements, but also on accumulating knowledge. Yu defined two methods of accumulation the knowledge: vertical knowledge accumulating method – when knowledge is accumulated with a person or a group; and horizontal knowledge accumulating method – when knowledge is accumulated by exchanging information between different persons and groups. Knowledge accumulation approach merged with requirement specification can provide effective and reliable way for designing test cases during acceptance testing. From the other hand this approach is difficult, because of the lack of the time and people, especially if software system is web-based, because it is hard to define target users for such kind of application.

One of the problems in acceptance testing is concerned with testing web applications. The reason is that there is no model for performing acceptance testing for web applications, which would specify how it should be performed. Traditional types of acceptance testing, such as alpha and beta testing does not consider the main advantages of web applications – acceptance testing for web applications can be performed remotely.

Yu proposed new testing model for performing user acceptance testing called Call for Testing (CFT). CFT guarantees the quality of acceptance testing providing the platform that constructs coverage criteria based on the user requirements and design documents, monitors community testers activities in real-time and verifies their performances (Yu et al, 2009).

CFT model propose using three roles:

- WebApp requirements Owner
- WebApp Developer
- WebApp Tester

Figure 4.5.1 provides collaboration process of CFT model.

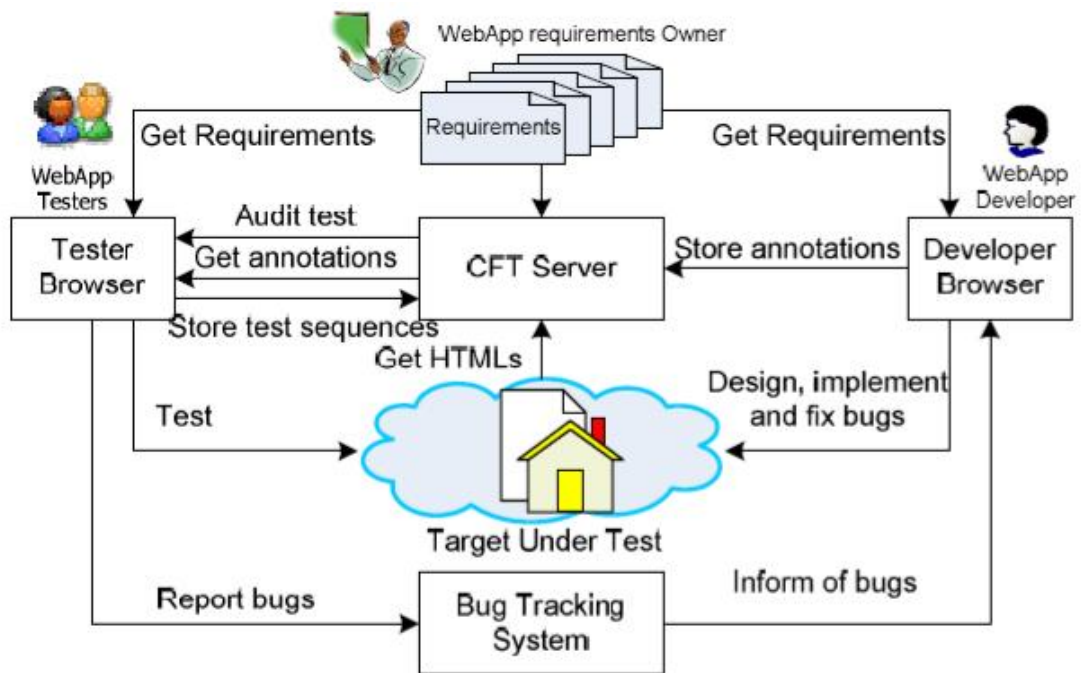


Figure 4.5.1 Collaboration process of CFT model (Yu et al, 2009).

Requirements for the system are provided by WebApp requirements owner.

Developers can specify the types of Web UI elements, define validation points for them, store this information in annotations and send it to SFT server. Developer receives information about requirements from requirement owner and information about bugs that should be fixed from Bug Tracking System. Tester can receive information about UI elements, getting annotations as well as perform testing activities and store information about test sequences that were performed. Test auditing is supported as follows: annotations, provided by developers, are used for creating test requirements, testing activities are parsed, and results are compared with testing requirements.

The whole architecture of CFT model is provided on Figure 4.5.2

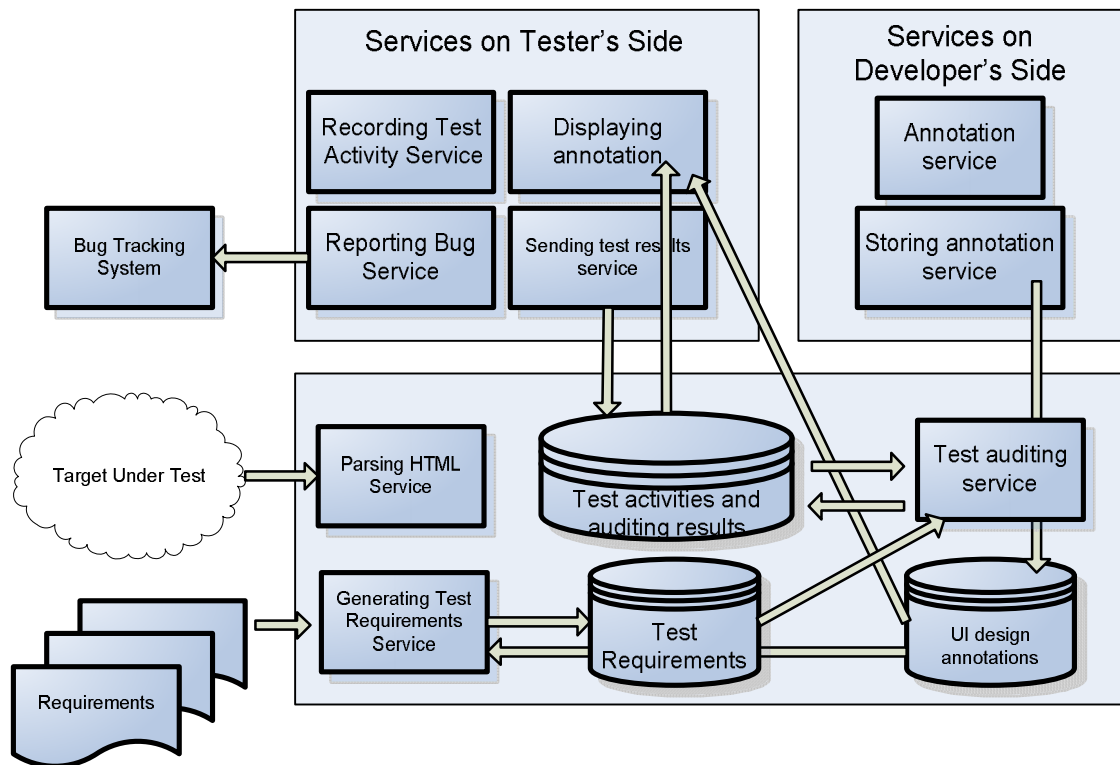


Figure 4.5.2 Architecture of CFT model (Yu et al, 2009).

Architecture of CFT model consists of three parts:

1. Tester's Services on client side (browser)

Tester services include displaying information about UI elements from annotations, provided by developers, getting information about not-tested UIs, facilities for recording test results and found bugs.

2. Developer's Services on client side (browser)

Developer has services that helps to work with annotations and store this information on CFT server. Annotation as was early said contain information about UI elements that should be validated and about the way how they should be validated.

3. CFT Server's Services

Services on CFT server side provide storage and audition facilities – store information about requirements, annotations and test results; compare testing requirements with testing results.

Call for Testing model provides facilities for performing user acceptance testing for Web-applications. CFT leverages the open community resource to contribute to Web testing, and imposes a lot of challenges to implement the model (Yu et al, 2009). From the other hand, CFT model requires both distributed testing infrastructure and a mechanism of testing audit to assess the testing quality that makes implementing of the model rather complex.

The main benefits of CFT:

1. Internet connection availability is the only condition for participating in UAT, so there's no limit of user's amount
2. CFT let users create test cases and perform UAT without interference of system's developers
3. Reducing testing time providing abilities for concurrent communication between testers and developers.

## 4.6 Usability testing

Usability testing focuses on measuring usability of the software involving target users in the testing process. Usability testing approach is based on considering user's interests and allows increasing user attraction that makes it important for the business.

Applying usability testing can help in reaching following profits:

- increasing customer's satisfaction
- increasing the number of software's users (new customers)
- increasing the profit of software (f.e. the number of customers for web site)
- decreasing maintenance effort
- decreasing education time, provided to customers

Meiyu mentioned the value of usability test is that the problems of tested product or service can be found early and improving advice can be given before the final decision of the product (Meiyu et al, 2008). Besides usability testing is valuable not only for problem identification, but also for competitive evaluations, and collecting quantitative data about a product's usability (Rosenbaum, 2007).

Usability testing suggests providing interviews with target users and discovering the inefficiencies and lacks of the user interface. Performing usability testing of developed interface only can face with problems that users are not satisfied with the interface. The reason is that usability testing is rather complex process that should be performed in several stages. There are approaches for testing usability before building software prototype. The instrument for testing is a walkthrough, which is a systematic review of a design on a paper. Traditional usability walkthrough allows providing the sequence of graphical elements, finding out is this sequence logic, how clear is this sequence for the user and is it consistent or not. Bias offers changed usability walkthrough in order to improve testing efficiency.

Improved walkthrough has following characteristics:

1. Three types of people should participate in walkthrough:
  - a. target users
  - b. software developers
  - c. human factors professional;
2. User interfaces should be presented in the order as they would appear in the application;
3. Each participant writes his comments to the every presented interface before any discussions.

The order of presenting information about interfaces is as follows: first – developers, second – target users, third – human factor professionals.

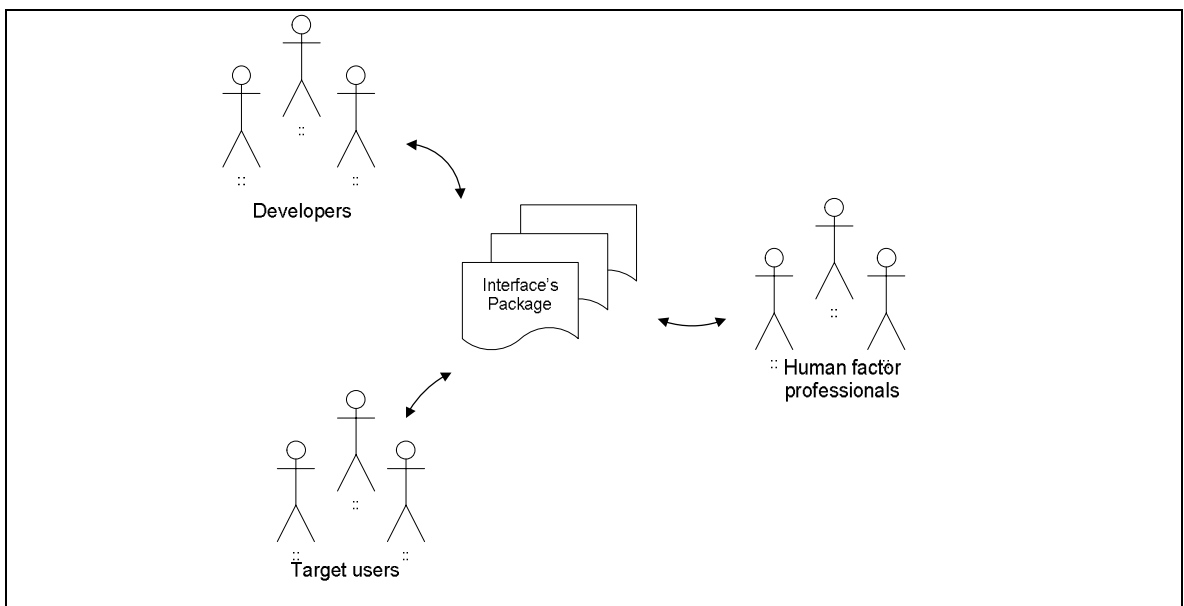


Figure 4.6.1 Usability Walkthrough Schema.

Figure 4.6.1 shows the schema of usability walkthrough. For each task, which should be explored during walkthrough, scenario should be defined. Scenario contains evaluating several steps, user interfaces participated in scenario are presented each on the separate

page in the order of their appearance. Each of the users gets the package of interface pictures for every defined scenario.

Target users are asked to write their actions for evaluating scenario. Information about these actions should be written in details, instead of writing 'Select item 3', users are encouraged to write 'Push the down arrow key two times, and then push Enter' (Bias, 1991). Writing particular actions will help to design the interface which will be understandable and expectable for the user.

Walkthrough is group process: after all the users presented their actions for each of the interfaces, the results are discussed. Human factor professionals play administrative role – they help express user's comments as cogent suggestions and guide the developers toward a particular usability improvement (Bias, 1991). The presence of human factor professionals in walkthrough's process allows to avoid misunderstanding between users and developers and to perform testing effectively.

The main benefit of this method is that during usability walkthrough, developers get written instructions from the users for each of the interfaces - what and how user will accomplish the task. The importance of group work is also stressed – according to Bias, applying this method helps to get valuable design data that one tend not to get when performing testing with users individually.



## **5. RESULTS AND DISCUSSION**

The initial objectives for this thesis were to provide a sufficient literature review on different testing phases and for each of the phases define the method that can be effectively used for improving software's quality. Software testing phases, chosen for study are: unit testing, integration testing, functional testing, system testing, acceptance testing and usability testing.

The research on software testing methods showed that there are many software testing methods that can be applied at different phases and in the most of the cases the choice of the method should be done depending on software type and its specification. The information about software testing phases and methods was presented in the following manner: the problem, concerned to each of the phases was identified; the method that can help in eliminating this problem was suggested and particularly described.

### **5.1 Discussion on suggested methods**

The discussion of suggested methods is performed in the order of their appearance in section 5: first method for discussion was suggested to be applied in the unit testing phase, then – integration testing, functional testing, system testing, acceptance testing and usability testing.

The problem, identified in the unit testing phase is that programming code view is not suitable for developing test cases: it is difficult both to build all necessary routes and create unique test cases. The decision of this problem is using graphical representation of the module, i.e. programming code should be converted to the graph. The main idea of suggested method is to make a representation of textual view using Coloured Petri Nets.

This method allows to get graphical structure of the programming code and to perform test case on the paper firstly. Comparing to other graphical representations of program's textual

view, such as more simple representation based on the binary tree, this method allows to build dynamic graphical representation, as it manipulates developed representation as a Petri Net.

The problem, identified for integration testing phase was concerned with object-oriented programming and consisted in that traditional methods are not appropriate for object-oriented programming. The reason is that object, which is the module in object-oriented programming is more than a programming module in common sense, and traditional integration testing methods does not take into account the structure of OO programs; so performing integration testing using these methods is difficult and inefficient. Suggested method allows generating test cases for integration testing from UML diagrams. The test cases for this method are assumed to consist of three parts: testing that the sequence of message calls conforms to the relevant sequence diagram; testing parameters; and testing object interactions by examining the states of objects after execution of prescribed sequences. The main profit of this method comparing to other methods of testing OO programs is that test cases are build using developed UML diagrams. That allows generating test cases according to the way how program was designed.

In functional testing the lack of traditional methods is that during the testing interior structure of the program is ignored and the only function that is considered while developing test cases is function evaluated by the program. Sometimes in program occur situations, when test case is passed successfully, but some variables or objects were set to wrong states, because some functions were incorrect. Suggested method requires developing the complete set of functional tests for each of the functions participating in program design – “design functions”. Each of design functions should be tested then. Design function testing can help to find errors, which can be skipped by black-testing methods, f.e. program with incorrect control function can work in a proper way, but not very efficiently.

There are different methods that can be used during system testing. In many respects these methods depends on software type, but there are also methods that can be used for wide range of software systems. Suggested method performs system testing based on UML. It uses sequence and statechart diagrams for generating test cases. The main challenge in system testing is coverage of the system states, because the number of system states is usually very big and system state model usually is not constructed. Comparing to other methods of system testing, suggested method covers all system states that can be reached from UML diagrams, that makes applying these method reasonable.

One of the problems in acceptance testing is concerned with testing web applications. Traditional types of acceptance testing, such as alpha and beta testing does not consider the main advantages of web applications – acceptance testing for web applications can be performed remotely. Proposed testing model allows performing acceptance testing remotely, besides it guarantees the quality of acceptance testing providing the platform that constructs coverage criteria based on the user requirements and design documents, monitors community testers activities in real-time and verifies their performances.

Performing usability testing of developed interface only can face with problems that users are not satisfied with the interface. The reason is that usability testing is rather complex process that should be performed in several stages. There are approaches for testing usability before building software prototype. The instruments that are usually applied for testing are walkthroughs, which are systematic reviews of a design on a paper. Comparing to traditional walkthroughs, suggested method is a modified usability walkthrough, which provides the instructions how to improve testing efficiency.

## 6. CONCLUSION

For this thesis literature review was conducted in order to define testing methods that can be used by developing team for improving quality of software. First two parts of the thesis are focused on software testing discipline: principles, testing methods and tools. Third part presents different phases of software testing and performed information why are they needed, how do they impact on the quality of software and what techniques are usually used during appropriate phase. Fourth part of thesis presents results of research on software testing methods – it provides one method for each of the testing phases, which can be effectively used for improving software's quality.

This thesis considered testing process as a process of validation and verification activities: validation helps to answer the question does one develop right product and verification - does one develop product right. The thesis concentrated on the study of validation activities that can be performed during the software development cycle.

The objectives for this thesis were to provide a literature review on different testing phases and for each of the phases define the method that can be effectively used for improving software's quality. Software testing phases, chosen for study are: unit testing, integration testing, functional testing, system testing, acceptance testing and usability testing. For each of the phases the problem that can occur was identified. Suggested methods were elicited in order to solve identified problem and improve software's quality. The discussion of methods was provided in section 6.

The thesis was done as a part of a software testing and quality assurance research project MASTO at Lappeenranta University of Technology. Methods discussed in thesis were suggested to the standardization group ISO/IEC JTC1/SC7 WG4 as methods to be included in new software testing standard ISO 29119. The perspectives for future work includes the continuation of identifying problems, concerned with testing process and the search of approaches that can help to solve these problems.

## REFERENCES

- Andrade, L.F., Fiadeiro, J.L.(1999), *Interconnecting Objects Via Contracts*, Robert France and Bernhard Rumpe (Eds.): <<UML>>'99, LNCS 1723, pp. 566-583, Springer-Verlag Berlin Heidelberg
- Cardoso, J., Camargo, H.(1999). *Fuzziness in Petri Nets. Studies in fuzziness and soft computing* Vol 22, Physica-Verlag Heidelberg, Germany.
- Cordic, V. (2008). *Petri Net. Theory and Applications*, I-Tech Education, Vienna, Austria.
- Developer.com, 2010. Available online at <http://www.developer.com>. Accessed 30.04.2010
- Dustin, E., Rashka, J., Paul, J.(2000). *Automated Software Testing*, ADDISON-WESLEY, England.
- Ellims, M., Bridges, J., Ince, D.C.(2004), *Unit testing in practice*, 15-th International Symposium on Software Reliability Engineering.
- Fagan, M.E.(1976), *Design and Code Inspections to Reduce Errors in Program Development*, IBM Systems J., 1976
- Hsia, P., Gao, J., Samuel, J., D. Kung, Toyoshima, Y., Chen, C.(1994) *Behavior-Based Acceptance Testing of Software Systems: A Formal Scenario Approach*, Computer Software and Applications Conference.
- IEEE/ANSI (1990), *IEEE Standard Glossary of Software Engineering Terminology*, 610.12-1990.
- Ipate, F., Lefticaru, R., *State-base testing is functional testing!*, Testing: Academic and Industrial Conference – Practice and Research Techniques.
- ISO 13407:1999, *Human-centred design processes for interactive systems*
- Bias, R.(1991) *Walkthroughs: efficient collaborative testing*, Aldus Corp, Seattle, WA.
- Kit, E. (1996). *Software Testing in the real world: improving the process*, ADDISON-WESLEY, England.
- Leung, H., K., N., White, L.(1990) *A Study of Integration Testing and Software Regression at the Integration Level*, Conference on Software Maintenance, 1990.

- Li ,Z., Maibaum, T. (2007) *An approach to integration testing of object-oriented programs*, Seventh International Conference on Software Quality.
- Meiyu, Lv., Wenjun, H., Chunjing Zhao(2008) *Research of usability test mode based on the implicit user behavior lib*, 9<sup>th</sup> International Conference on Computer-aided industrial design and conceptual design, CAID/CD, 2008.
- Myers, G. (2004).*The Art of Software Testing*, John Wiley & Sons, Hoboken, New Jersey, USA
- Omar, A.A., Mohammed (1991), *A survey of functional software testing methods*, ACM Sigsoft, Software Engineering Notes vol.16 no.2
- Roe, R.P., Rowland, J.H.(1987), *Some theory concerning certification of mathematical subroutines by black box testing*, IEEE Transactions on Software Engineering, vol. SE-13, no.6, June 1987
- Rosenbaum, S., Kantner, L.(2007), *Field usability testing: method, not compromise*, Professional Communication Conference, 2007. IPCC 2007, IEEE International.
- Sarba, M., Mall, R.(2007) , *System testing using UML Models*, 16<sup>th</sup> IEEE Asian Test Symposium.
- Shildt, H.(2003) *C++: The complete reference. Forth Edition*, The McGraw-Hill Companies Inc, Beijing, China.
- Software Development Technologies. Available online at <http://www.sdtcorp.com/> 30.04.2010
- Taipale, O.(2007); *Observations on software Testing Practice*; Doctor of science thesis; Lappeenranta University of Technology, 2007.
- Zhang, N., Bao, X., Ding, Z. (2009). *Unit Testing: Static Analysis and Dynamic Analysis*, Fourth International Conference on Computer Sciences and Convergence Information Technology.
- Yu, Y., Wu, F.(1999) *A software acceptance testing technique based on knowledge accumulation*, Proceedings. Ninth Great Lake Symposium on VLSI, 1999