Test plan

Author: Eric Åberg

Version: 1.0

Date: 2005-11-11

Summary

This document describes the planning of all the test phases of the project 'Wazzup DOC?!' that PUM group 12 is developing in the course TDDC02 at Linköping University.

It also describes the test specification of all the test phases such as test cases and test order.

Responsibilities and time planning are brought up as well.

Project identity

Project group

PUM 12 2005
Linköpings tekniska högskola
Institutionen för datavetenskap (IDA)

Project members

Name	Responsibility	Phone number	E-mail
Martin Jormedal	Project leader (PL)	073-3121319	ook4mi@gmail.com
Daniel Hilding	Customer relations (CRM)	070-7440440	danhi139@student.liu.se
Joakim Svartengren	Documentation manager (DOC)	070-4040005	joasv190@student.liu.se
Jonas Norling	Design manager (DES)	070-3904809	norling@lysator.liu.se
Johan Lissing	Implementation manager (IM)	073-9036256	johli650@student.liu.se
Thobias Bergqvist	Quality manager (QM)	073-6223040	thobe651@student.liu.se
Eric Åberg	Test manager (TM)	070-4058130	eriab522@student.liu.se

Mailinglist for group

pum12@und.ida.liu.se

Web page

http://www-und.ida.liu.se/~pum12/

Customer

Per Karlström, ISY LiU

Customer contact person

Per Karlström, 013-28 29 03, perk@isy.liu.se

Project supervisor

David Broman, 013-28 57 24, davbr@ida.liu.se

Examiner

Robert Kaminski, 013-28 24 57, robka@ida.liu.se

Document History

Date	Version	Changes	Name
2005-11-05	0.1	Document created	Eric Åberg
2005-11-08	0.2	Fixed spelling errors after commenting. Added cross-references and references. Added the summary.	Eric Åberg
2005-11-11	1.0	Fixed spelling errors after inspection. Changed the structure of the module specification slightly. Added two tests in the module specification.	Eric Åberg

1 Introduction 11
1.1 Purpose 11
1.2 Philosophy 11
1.3 Chapter overview 11
1.3.1 Introduction 11
1.3.2 Global time plan 11
1.3.3 Unit test plan 11
1.3.4 Module test plan 11
1.3.5 Integration test plan 11
1.3.6 System test plan 12
1.3.7 Acceptance test plan 12
1.3.8 Module test specification 12
1.3.9 Integration test specification 12
1.3.10 System test specification 12
1.3.11 Acceptance test specification 12
1.4 Reading instructions 12
1.5 Document dependencies 12
1.6 Distribution 12
1.7 Glossary 13
2 Global time plan 15
2.1 Time plan 15
2.2 Test order 15
3 Unit test plan 17
3.1 Goals 17
3.2 Method 17
4 Module test plan 19
4.1 Goals 19
4.2 Personnel and responsibiliteies 19
4.2.1 Test manager 19
4.2.2 Test constructor 19
4.2.3 Test secretary 19
4.2.4 Tester 19
4.3 Time plan 20
4.4 Methods 20
4.4.1 Static code inspection 20
4.4.2 Black box testing 20
4.5 Criteria before starting tests 20
4.6 Criteria for successful tests 20
4.7 Equipment and tools 21
4.7.1 Equipment 21

4.7.2 Documents 21
4.8 Test procedure 21
4.8.1 Static code inspection 21
4.8.2 Black box testing 22

5 Integration test plan 23
5.1 Goals 23
5.2 Personnel and responsibilities 23
5.2.1 Test manager 23
5.2.2 Test constructor 23
5.2.3 Test secretary 23
5.2.4 Tester 23
5.3 Time plan 24
5.4 Methods 24
5.5 Criteria before starting tests 24
5.6 Criteria for successful tests 24
5.7 Equipment and tools 25
5.7.1 Equipment 25
5.7.2 Documents 25
5.8 Test procedure 25
5.8.1 Preparations 25
5.8.2 Execution 25
5.8.3 Follow-up 25

6 System test plan 27
6.1 Goals 27
6.2 Personnel and responsibilities 27
6.2.1 Test manager 27
6.2.2 Test constructor 27
6.2.3 Test secretary 27
6.2.4 Tester 27
6.3 Time plan 28
6.4 Methods 28
6.5 Criteria before starting tests 28
6.6 Criteria for successful tests 28
6.7 Equipment and tools 28
6.7.1 Equipment 28
6.7.2 Documents 28
6.8 Test procedure 29
6.8.1 Preparations 29
6.8.2 Execution 29
6.8.3 Follow-up 29

7 Acceptance test plan 31
7.1 Goals 31
7.2 Personnel and responsibilities 31
7.2.1 Test manager 31
7.2.2 Test constructor 31
7.2.3 Test secretary 31
7.2.4 Tester 31
7.3 Time plan 32
7.4 Methods 32
7.5 Criteria before starting tests 32
7.6 Criteria for successful tests 32
7.7 Equipment and tools 32
7.7.1 Equipment 32
7.7.2 Documents 33
7.8 Test procedure 33
7.8.1 Preparations 33
7.8.2 Execution 33
7.8.3 Follow-up 33

8 Module test specification 35
8.1 Purpose 35
8.2 Description 35
8.3 Modules 35
8.3.1 Tree printer 35
8.3.2 Verilog front-end 35
8.4 Test order 35
8.5 Test cases 36
8.5.1 Tree printer 36
8.5.2 Verilog front-end 39

9 Integration test specification 45
9.1 Purpose 45
9.2 Description 45
9.3 Test modules 45

10 System test specification 47
10.1 Purpose 47
10.2 Description 47
10.3 Test order 47
10.4 Account for satisfactory system test 47
10.5 Test cases. 48

11 Acceptance test specification 51
11.1 Purpose 51
11.2 Description 51
11.3 Test order 51
11.4 Account for satisfactory system test 51
11.5 Test cases 51
12 References 53
12.1 Internal documents 53
12.2 External documents 53

1 Introduction

This chapter briefly describes what the document is about, how to comprehend it the best way and which documents are influenced by it.

1.1 Purpose

The purpose of this document is to allow the testing go as quick and easy as possible and to ensure that the tests do not start until everyting is prepared. It is foremost written to make sure that the final product gets the best possible quality.

1.2 Philosophy

Since a part of this project is to investigate if it is possible to extend Doxygen to document Verilog code, there is a problem which modules and classes that need to be tested. Our basic requirements cover the parser and the tree printer. These are the only classes with methods we know that at the present time are known to be created. Therefore these are the only ones we can test. If there are other modules that need testing, they will be covered in section 4 "Module test plan" and section 5 "Integration test plan" .

Additional test cases need to be created in section 5 "Integration test plan" and section 9 "Integration test specification" if this should happen.

1.3 Chapter overview

1.3.1 Introduction

This chapter briefly describes what the document is about, how to comprehend it the best way and which documents are influenced by it.

1.3.2 Global time plan

This chapter contains a global time plan of the testing in the project as well as the order the testing.

1.3.3 Unit test plan

This chapter contains the goals of the unit testing and the method we are using.

1.3.4 Module test plan

This chapter describes by which group members the module testing will be performed, how much time the tests can use, and criteria which must be fulfilled before a test can begin and before the group can call the test a success.

1.3.5 Integration test plan

This chapter describes by which group members the integration testing will be performed. How much time the tests can take and criteries which must be fulfilled before a test can begin and before the group can call the test a success.

1.3.6 System test plan

This chapter contains the goals of the system test, Which personel that are involved and how the test is performed. It also contains a time plan for the system test.

1.3.7 Acceptance test plan

This chapter contains the goals of the acceptance test, Which personel that are involved and how the test is performed. It also contains a time plan for the acceptance test.

1.3.8 Module test specification

This chapter contains the test cases used in the module test and in which order they will be executed.

1.3.9 Integration test specification

This chapter contains the some information of at the present non-existing integration test.

1.3.10 System test specification

This chapter contains the test cases used in the system test and in which order the tests will be executed.

1.3.11 Acceptance test specification

This chapter contains a reference to the test cases for the acceptance test and the order for them to be executed.

1.4 Reading instructions

To get an overall understanding of the planning of the testing the section 2 "Global time plan" should be read.

The planning of the tests of the different phases is described in section 3 "Unit test plan" to section 6 "System test plan"

The test cases for every phase is described in section 8 "Module test specification" to section 11 "Acceptance test specification" .

Detailed test scripts will be written during the test phase and will be published in the Test report [Åberg, 2005] as well as the test results.

1.5 Document dependencies

This document is dependent of the following docuements:

- Requirements specification [Hilding, 2005]
- Architecture specification [Norling, 2005]
- Design specification [Lissing, 2005:2]
- Project plan [Jormedal, 2005]

1.6 Distribution

This document will be distributed to:

- Document examinors: Peter Bunus and Stina Edelfeldt.
- Our supervior David Broman.
- The project folder.
- The home page the project.

1.7 Glossary

AST - Abstract Syntax Tree. A tree describing the structure of a source file.

Parser - A piece of software that determines the syntactic structure of a language.

Stubs - Code that makes a module or several modules run wihout errors from other missing modules.

2 Global time plan

This chapter contains a global time plan of the testing in the project as well as the order of the testing.

2.1 Time plan

This is the time plan for the testing related work during the project. A more detailed time plan is placed in every testing plan..

Week	Phase	Event	Persons
35	Definition phase	Test plan in the quality report	TM
37	Definition phase	Acceptance test in the requirement specification	TM
44	Design phase	Test plan	TM
46	Testing phase	Test script creation	TM, DOC, CRM
46	Testing phase	Module testing	TM, DOC, CRM
48	Testing phase	Integration testing	TM, PL
49	Testing phase	System testing	TM, PL
50	Testing phase	Acceptance testing	TM, PL, CRM
49	Testing phase	Test report	TM, PL, CRM, DOC

Table 13: Table over the global time plan.

2.2 Test order

First of all the tree printer module has to be tested. When it is approved, the testing can enter a module/integration test phase, where the tree printer module is tested together with the Verilog front-end module.

The system test will take place when all the integration testing is complete. The product is then ready for the customer to approve in the acceptance test.

3 Unit test plan

This chapter contains the goals of the unit testing and the method we are using.

3.1 Goals

The goal with unit testing is to avoid those small errors (i.e. spelling errors and mistakes made by the programmer) that always exist in newly written code.

3.2 Method

The method we are using to handle this is the most common one: The programmer himself tests the function when a new one is written. We feel that this is the best and most time conserving method at this state of the project.

4 Module test plan

This chapter describes by which group members the module testing will be performed, how much time the tests may use, and criteria which must be fulfilled before a test can begin and before the group can call the test a success.

4.1 Goals

The goal with module testing is to find as many bugs and errors as possible in the inner functions of the modules.

4.2 Personnel and responsibiliteies

Each test needs a group of personnel with different responsibilities. One of each of the following roles is needed.

4.2.1 Test manager

The test manager has got the main responsibility of the module testing.

The test manager makes sure that everyone in the test group knows their roles during the module test.

The test manager has got the responsibility to make sure that every module follows the test specification.

The test manager will evaluate the test reports and decide if further testing is needed.

4.2.2 Test constructor

The test constructor creates the test script together with the test input and makes sure it is runnable.

The test constructor creates a test protocol for the test secretary to fill in.

4.2.3 Test secretary

Every test has a test secretary who writes a test protocol. Every bug and error is recorded in the protocol, so it is easy to findit later if needed. The secretary role is given to the code writer of the module.

4.2.4 Tester

The tester executes the script together with the programmer.

4.3 Time plan

Person	Responsibility	Task	Available time
TM	Responsible of all testing	Administration	9 h
CRM	Test constructor	Constructs tests	3 h
DOC	Test constructor	Constructs tests	7 h
TM	Tester	Executes the tests	3 h
DOC	Tester	Executes the tests	3 h
CRM	Tester	Executes the tests	2 h

Table 14: Time plan for module testing

4.4 Methods

There are two kinds of test steps, Code inspection and Program execution. Since the project has not got nearly enough time to test everything, the test group will perform two kind of tests. These methods are stated below.

4.4.1 Static code inspection

All written code must be controlled before the testing of module begins. It is important that the code follows the the standard written in Programming handbook [Lissing, 2005:1] e.g. File naming, indentation, commenting according to doxygen standard etc.

4.4.2 Black box testing

When the static code inspection has been passed, the black box testing begins. The main purpose of black box testing method is that the test constructor knows nothing about how the module is written. The only thing he knows is how the output data should look like depending on the input data.

4.5 Criteria before starting tests

To make sure that the test goes smoothly and the time is well spent, there are some points that have to be checked before starting a test. Every point below has to be fulfilled before the test can start.

- Test script and input are ready.
- Test protocol is ready.
- Test members, know where and when the test takes place.

4.6 Criteria for successful tests

A specific test case passes if the result is equivalent to the expected result. A module test passes when an acceptable number of the test cases have been

passed. The entire module test phase passes when all modules have been passed.

The testing can not test everything in the modules and even if it could, everyting can not be expected to be corrected. The test manager has to consider the quantity of the errors together with the size of the errors. If the test manager is not happy with the result, the test manager requests another test when the errors have been corrected.

When the test manager feels that the testing is done the test manager evaluates the results together with the quality manager and the project leader.

4.7 Equipment and tools

4.7.1 Equipment

- PC with linux/unix OS
- The current module

4.7.2 Documents

- Module test plan
- Module test specification
- Test scripts
- Test protocol
- Error reports
- Programming handbook[Lissing, 2005:1]

4.8 Test procedure

This section contains the performance of the module testing.

4.8.1 Static code inspection

This section contains the performance of the static code inspection. Before the static code inspection can begin there are some preparations that have to be performed. When the test is performed the follow-up begins. The follow-up is performed so the errors can be corrected and it is easy to look back on the errors in the future.

4.8.1.1 Preparations

When the code writer feels that a module is ready, he informs the test manager about it. The test manager summons the test group and makes sure that all the participants have read the Programming handbook [Lissing, 2005:1] and the checklist by the test secretary.

4.8.1.2 Execution

The tester reads through the code and checks the checklist. If there is an error it is recorded in the protocol.

4.8.1.3 Follow-up

The results are placed on the project homepage and in the project folder. The results are then discussed according to section 4.6 "Criteria for successful tests" . When the static code inspection has passed, the test manager starts the blackbox testing.

4.8.2 Black box testing

This section contains the performance of black box testing. Before black box testing can begin there are some preparations thats have to be performed. When the test is performed the follow-up begins. The follow-up is performed so the errors can be corrected and it is easy to look back on the errors in the future.

4.8.2.1 Preparation

The test constructor writes the test script and all the belonging code, such as stubs, according to the test specification, so that the test is runnable without errors from other missing modules. The tester reads the module test plan and module test specification. The static code inspection must have been finished before the execution och the black box testing begins. All Verilog files that are used as inputs must be synthesizable before they can be run in the test.

4.8.2.2 Execution

The test construcor provides a computer with the module, test script and the belonging code to the tester. The tester executes the test scripts and the code writer of the module records the results.

4.8.2.3 Follow-up

The results are placed on the project homepage and in the project folder. The results are then discussed according to section 4.6 "Criteria for successful tests" .

If the module has been through a lot of retests the module’s design might be questionable. It might be a better idea to change the design of the module rather than keep testing it.

If the module passes at once without any errors at all or with very few errors there might be someting wrong with the test script.

The Test manager will decide what measures that will be taken if any of these cases should appear.

5 Integration test plan

This chapter describes by which group members the integration testing will be performed. How much time the tests may take, and criteria which must be fulfilled before a test can begin and before the group can call the test a success. Since there are only two existing modules at the moment and they are integrated in the module test, there will be no testing in this phase yet.

5.1 Goals

The goal with integration testing is to find as many bugs and errors as possible in the interface between modules.

5.2 Personnel and responsibilities

Each test needs a group of personnel with different responsibilities. One of each of the following roles is needed.

5.2.1 Test manager

The test manager has got the main responsibility of the integration testing.

The test manager makes sure that everyone in the test group knows their roles during the integration test.

The test manager has got the responsibility to make sure that the integration follows the test specification.

The test manager will evaluate the test reports and decide if further testing is needed.

5.2.2 Test constructor

The test constructor creates the test script together with the test input and makes sure it is runnable.

The test constructor creates a test protocol for the test secretary to fill in.

5.2.3 Test secretary

Every test has a test secretary who writes a test protocol. Every bug and error is recorded in the protocol, so it is easy to findit later if needed. The secretary role is given to one of the code writers of the modules.

5.2.4 Tester

The tester executes the script together with the programmer.

5.3 Time plan

Person	Responsibility	Task	Available time
TM	Responsible of all testing	Administration	5 h
TM	Test constructor	Constructs tests	5 h
PR	Tester	Executes the tests	3 h

Table 15: Time plan for integration testing

5.4 Methods

Since the module testing and integration testing is intermixed at the moment, we have chosen to test according to the functional test in "RUT - Development Handbook 12.1 Choice of Test Strategy v6.0" [Arvedahl, 2002]. This method is a lot like the module testing's 'black box testing' which we used in the the module testing.

There are some indications in the Design specification[Lissing, 2005:2] that there will be changes in the Data organizer module as well as the some other modules during the future investigation and implementation. If this is the case there might be changes to the integration test plan when it is in question.

5.5 Criteria before starting tests

To make sure that the test goes smoothly and the time is well spent there are some points that have to be checked before starting a test. Every point below has to be fulfilled before the test can start.

- Test script and input are ready.
- Test protocol is ready.
- Test members knows where and when the test takes place.

5.6 Criteria for successful tests

A specific test case passes if the result is equivalent to the expected result. The entire integration test passes when all of the test cases have been passed.

The testing can not test everything in the modules and even if it could, everyting can not be expected to be corrected. The test manager have to consider the quantity of the errors together with the size of the errors. If the test manager is not happy with the result, the test manager requests another test when the errors have been corrected.

When the test manager feels that the testing is done, the test manager evaluates the results together with the quality manager and the project leader.

5.7 Equipment and tools

5.7.1 Equipment

- PC with linux/unix OS
- The current modules

5.7.2 Documents

- Integration test plan
- Integration test specification
- Test scripts
- Test protocol
- Error reports

5.8 Test procedure

This section contains the performance of the module testing.

5.8.1 Preparations

The test manager makes sure that the test specification for the system test is available. The whole test group should have read this in an early stage of the test period. The test constructor has written test scripts to match the specification and is sure that all the tests are runnable. The test constructor also makes sure that everything needed is provided to the tester i.e. computer, checklist, testscripts. Time and place is announced by the test manager. All Verilog files that are used as inputs must be synthesizeable before they can be run in the test.

5.8.2 Execution

The tester and the test secretary conduct all of the tests. The result is recorded in the protocol by the test secretary.

5.8.3 Follow-up

The results are placed on the project homepage and in the project folder. The results are then discussed according to section 5.6 "Criteria for successful tests" .

6 System test plan

This chapter contains the goals of the system test, which personnel that are involved and how the test is performed. It also contains a time plan for the system test.

6.1 Goals

The goal with the system test is to find differences between the existing product and the Requirements specification [Hilding, 2005].

6.2 Personnel and responsibilities

6.2.1 Test manager

The test manager has got the main responsibility of the system testing.

The test manager makes sure that everyone in the test group knows their roles during the system test.

The test manager has got the responsibility to make sure that every system follows the test specification.

The test manager will evaluate the test reports and decide if further testing is needed.

6.2.2 Test constructor

The test constructor creates the test script together with the inputs and makes sure it is runnable.

The test constructor creates a test protocol for the test secretary to fill in.

6.2.3 Test secretary

Every test has a test secretary who writes a test protocol. Every bug and error is recorded in the protocol, so it is easy to findit later if needed. The secretary role is given to the code writers of the system.

6.2.4 Tester

The tester executes the script together with the programmer.

6.3 Time plan

Person	Responsibility	Task	Available time
Test manager	Responsible of all testing	Administration	7 h
Test manager	Test constructor	Constructs tests	5 h
Project leader	Tester	Executes the tests	2 h

6.4 Methods

- Test the system according to the Requirements specification [Hilding, 2005].
- Test the performance of the system.

6.5 Criteria before starting tests

- The integration test is completed.
- System test specification and system test scripts are ready.
- Test members knows where and when the test takes place.

6.6 Criteria for successful tests

A specific test case passes if the result is equivalent to the expected result. The system passes when all of the test cases have been passed.

The testing can not test everything in the System and even if it could, everything can not be expected to be corrected. The test manager has to consider the quantity of the errors together with the size of the errors. If the test manager is not happy with the result, the test manager requests another test after the errors have been corrected.

When the test manager feels that the testing is done, the test manager evaluates the results together with the quality manager and the project leader.

6.7 Equipment and tools

6.7.1 Equipment

- PC with linux/unix OS
- The system.

6.7.2 Documents

- System test plan
- System test specification
- Test scripts
- Test protocol
- Error reports

6.8 Test procedure

6.8.1 Preparations

The test manager makes sure that the test specifiction for the system test is available. The whole test group should have read this in an early state of the test period. The test constructor has written test scripts to match the specification and is sure that all the tests are runnable. The test constructor also makes sure that everything needed is provided to the tester i.e. computer, checklist, testscripts. Time and place are announced by the test manager. All Verilog files that are used as inputs must be synthesizeable before they can be run in the test.

6.8.2 Execution

The tester and the test secretary conduct all of the tests. The result is recorded in the protocol by the secretary.

6.8.3 Follow-up

The test manager summons the entire test group when the system testing is complete. The group will discuss:

- Were the test cases relevant?
- Were all the system functions tested?
- Were all the requirements tested?

Errors that are discovered during the system test might lead to a renegotiation with the customer, since there is a great risk that the time it will take to fix the errors is too long.

7 Acceptance test plan

This chapter contains the goals of the acceptance test, Which personel that are involved and how the test is performed. It also contains a time plan for the acceptance test.

7.1 Goals

The goal with the acceptance test is to show the customer that the product fulfils the requirements in the Requirements specification [Hilding, 2005] .

7.2 Personnel and responsibilities

7.2.1 Test manager

The test manager has got the main responsibility of the acceptance testing.

The test manager makes sure that everyone in the test group knows their roles during the acceptance test.

7.2.2 Test constructor

The test constructor creates the test script together with the inputs and makes sure it is runnable.

The test constructor creates a test protocol for the test secretary to fill in.

7.2.3 Test secretary

Every test has a test secretary who writes a test protocol. Every design error is recorded in the protocol, so it is easy to find it later if needed. The secretary role is given to one of code writers of the system.

7.2.4 Tester

The tester executes the script together with the programmer.

7.3 Time plan

Person	Responsibility	Task	Available time
TM	Responsible of all testing and expertise.	Administration and answer questions from the customer.	5 h
CRM	Expertise	Answer questions from the customer.	4 h
DES	Expertise	Answer questions from the customer.	2 h
DOC	Expertise	Answer questions from the customer.	2 h
QM	Expertise	Answer questions from the customer.	2 h
CRM	Expertise	Answer questions from the customer.	2 h
IM	Expertise	Answer questions from the customer.	2 h

7.4 Methods

The method used during the acceptance test will be benchmark testing, as described in "RUT - development tutorial 14.1 Acceptance test v 2.0" [Karlsson, 2005]. This is to make it easy to prepare code examples for the customer to use.

7.5 Criteria before starting tests

- The system test is completed.
- Acceptance test specification and acceptance test scripts are ready.
- Test members know where and when the test takes place.

7.6 Criteria for successful tests

A specific test case passes if the result is equivalent to the expected result. The acceptance test passes when all of the test cases have been passed.

7.7 Equipment and tools

7.7.1 Equipment

- PC with linux/unix OS

- The system

7.7.2 Documents

- Acceptance test plan
- Acceptance test specification
- Test scripts
- Test protocol
- Error reports

7.8 Test procedure

7.8.1 Preparations

The test manager makes sure that the test specification for the acceptance test is available. The whole test group should have read this in an early stage of the test period. The test constructor has written test scripts to match the specification and is sure that all the tests are runnable. The test constructor also makes sure that everything needed is provided to the tester i.e. computer, checklist, test scripts. Time and place are announced by the test manager to the test group and to the customer. All Verilog files that are used as inputs must be synthesizeable before they can be run in the test.

7.8.2 Execution

The whole project group conduct all of the tests together with the customer. The result is recorded in the protocol by the secretary. The whole project group attend to make it easy to answer different kinds of questions from the customer.

7.8.3 Follow-up

When the test is completed, the group analyses and discusses the result together with the customer. If the customer is happy with the test, he signs the contract in the requirement specification. If the customer is not satisfied a solution has to be agreed upon.

8 Module test specification

This chapter contains the test cases used in the module test and in which order they will be executed.

8.1 Purpose

The purpose of this test is to find as many errors and bugs as possible in the modules. This is done to prevent errors from appearing later in the testing with a dreadful outcome. The test cases are written to make it easier for the test constructors to create the tests.

8.2 Description

Each test will be stated as below with a test id as MT-X where X increments with the tests.

Test id	Identification of the module test.
Purpose	Describes why the test is being done.
Reference	The class(-es) and function(-s) that will be tested.
Data input	The data that has to be inserted into the system to be able to carry out the test.
Execution	Describes how the test should be carried out.
Expected result	If the test result is the same as the expected result, the test has passed.

Test scripts and surrounding code will be created during the implementation phase.

8.3 Modules

These are the module that we know someting about at the present time in the project. More modules will be added as the investigation progresses.

8.3.1 Tree printer

This is the module that writes the syntax tree into a readable form.

8.3.2 Verilog front-end

Verilog front-end parses through the code and generates data structures which are very hard to interpret by a human without the tree printer. This is why we have integrated the tree printer into this test.

8.4 Test order

We will start with MT-1 and then perform the tests in increasing order.

The first module that will be tested is the Tree printer. When this is completed the test of the Verilog front-end module begins.

8.5 Test cases

8.5.1 Tree printer

These are the test cases for the Tree printer. The output for of the Tree printer will be compared to the definitions in the Architecture specification [Norling, 2005].

Test id	MT-1
Purpose	To make sure that it writes out every Verilog module that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST containing 3 modules.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing the 3 modules.

Test id	MT-2
Purpose	To make sure that it writes out every Verilog module argument that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST containing 3 modules with different arguments.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing the 3 modules with corresponding argument.

Test id	MT-3
Purpose	To make sure that it writes out a Doxygen comment that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST containing a comment block with some text.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing the comment, with the comment associated with the correct node.

Test id	MT-4
Purpose	To make sure that it writes out every wire that the AST contains.
Reference	TreePrinter::print(:Entry*).

Data input	An AST containing a module with 3 modules with 3 wires in each module.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing a module with 3 modules with 3 wires in each module.

Test id	MT-5
Purpose	To make sure that it writes out modules instantiated in other modules that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST containing 3 instantiated modules with 3 wires in each module.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing 3 instantiated modules with 3 wires in each module.

Test id	MT-6
Purpose	To make sure that it writes out the custom directive ’author’ that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST that contains a custom directive ’author’ in one module.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing an ’author’ in the module.

Test id	MT-7
Purpose	To make sure that it writes out the custom directive ’date’ that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST that contains a custom directive ’date’ in one module.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing an ’date’ in the module.

Test id	MT-8
Purpose	To make sure that it writes out the custom directive ’bug’ that the AST contains.

Reference	TreePrinter::print(:Entry*).
Data input	An AST that contains a custom directive 'bug' in one module.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing a 'bug' in the module.

Test id	MT-9
Purpose	To make sure that it writes out the custom directive 'clock' that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST that contains a custom directive 'clock' in one module.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing a 'clock' in the module.

Test id	MT-10
Purpose	To make sure that it writes out the custom directive 'reset' that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST that contains a custom directive 'reset' in one module.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing a 'reset' in the module.

Test id	MT-11
Purpose	To make sure that it writes out the custom directive 'comb' that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST that contains a custom directive 'comb' in one module.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing a 'comb' in the module.

Test id	MT-12
Purpose	To make sure that it writes out the custom directive 'state' that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST that contains a custom directive 'state' in one module.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing a 'state' in the module.

Test id	MT-13
Purpose	To make sure that it writes out the custom directive 'class' that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST that contains a custom directive 'class' in one module.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing a 'class' in the module.

Test id	MT-14
Purpose	To make sure that it writes out all the custom directives together with the modules and wires that the AST contains.
Reference	TreePrinter::print(:Entry*).
Data input	An AST that contains all the custom directives inside instantiated modules with wires in one module.
Execution	Execute the function print with the AST as input.
Expected result	A readable form of the AST containing all the custom directives, modules and wires listed above.

8.5.2 Verilog front-end

The Verilog front-end parses the file and generates an AST which is the input to Tree printer. The Tree printer then writes the AST as a text readable by a human. We will use a test bench, created by the group, which calls the

parseInput() in VerilogLnguageScanner with all the right arguments. The test bench needs a Verilog file as input.

Test id	MT-15
Purpose	To make sure that parser parses every module that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	An verilog file containing 3 modules.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing the 3 modules.

Test id	MT-16
Purpose	To make sure that the parser parses every module argument that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	An Verilog file containing 3 modules with different arguments.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing the 3 modules with corresponding argument.

Test id	MT-17
Purpose	To make sure that the parser parses every module argument that the Verilog files contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	Three Verilog files containing 3 modules in different files with different arguments.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing the 3 modules with corresponding argument.

Test id	MT-18
Purpose	To make sure that the parser parses an ordinary comment that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	An Verilog file containing a comment block with some text.

Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing the comment.

Test id	MT-19
Purpose	To make sure that the parser parses every wire that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	A Verilog file containing 3 modules with 3 wires in each module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing 3 modules with 3 wires in each module.

Test id	MT-20
Purpose	To make sure that the parser parses modules instantiated in other modules that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	A Verilog file containing 3 instantiated modules with 3 wires in each module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing 3 instantiated modules with 3 wires in each module.

Test id	MT-21
Purpose	To make sure that the parser parses the custom directive 'author' that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	A Verilog file that contains a custom directive 'author' in one module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing an 'author' in the module.

Test id	MT-22
Purpose	To make sure that the parser parses the custom directive 'date' that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)

Data input	A Verilog file that contains a custom directive 'date' in one module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing an 'date' in the module.

Test id	MT-23
Purpose	To make sure that the parser parses the custom directive 'bug' that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	A Verilog file that contains a custom directive 'bug' in one module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing a 'bug' in the module.

Test id	MT-24
Purpose	To make sure that the parser parses the custom directive 'clock' that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	A Verilog file that contains a custom directive 'clock' in one module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing a 'clock' in the module.

Test id	MT-25
Purpose	To make sure that parser parses the custom directive 'reset' that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	A Verilog file that contains a custom directive 'reset' in one module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing a 'reset' in the module.

Test id	MT-26
Purpose	To make sure that the parser parses the custom directive 'comb' that the Verilog file contains.

Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	A Verilog file that contains a custom directive 'comb' in one module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing a 'comb' in the module.

Test id	MT-27
Purpose	To make sure that the parser parses the custom directive 'state' that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	A Verilog file that contains a custom directive 'state' in one module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing a 'state' in the module.

Test id	MT-28
Purpose	To make sure that the parser parses the custom directive 'class' that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	A Verilog file that contains a custom directive 'class' in one module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing a 'class' in the module.

Test id	MT-29
Purpose	To make sure that the parser parses all the custom directives together with the modules and wires that the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	A Verilog file that contains all the custom directives inside instantiated modules with wires in one module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing all the custom directives, modules and wires listed above.

Test id	MT-30
Purpose	To make sure that the parser parses all the custom directives together with the modules in different files and wires that the Verilog files contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	Three Verilog files that containing all the custom directives inside instantiated modules in different files with wires in one module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file containing all the custom directives, modules and wires listed above.

Test id	MT-31
Purpose	To make sure that the parser ignores the custom directive 'clock' and 'reset' when they are declared with non-existing wires in the Verilog file contains.
Reference	VerilogLanguageScanner::parseInput(:Entry*)
Data input	A Verilog file that contains a custom directive 'clock' and 'reset' pointing to wires that do not exist in one module.
Execution	Execute the test bench with the Verilog file as input.
Expected result	A readable form of the Verilog file not containing a 'clock' or 'reset' in the module. A warning is generated.

9 Integration test specification

This chapter contains information of the presently non-existing integration test.

9.1 Purpose

The integration test is performed to make sure that every module interface works together with each other. This is to make it easier to locate an error therefore make it easier to correct. The test cases are written to make it easier for the test constructors to create the tests.

9.2 Description

Each test will be stated as below with a test id as IT-X where X increments with the tests..

Test id	Identification of the module test.
Purpose	Describes why the test is being done.
Reference	The class(-es) that will be tested.
Data input	The data that has to be inserted into the system to be able to carry out the test.
Execution	Describes how the test should be carried out.
Expected result	If the test result is the same as the expected result, the test has passed.

Test scripts and surrounding code will be created during the implementation phase.

9.3 Test modules

As it is now, there are no modules being tested in the integration test. One integration(Tree printer, Verilog front-end) took place during the module test. There will probably appear a number of tests as the project progresses. This will be handled at that time and test cases will be written.

10 System test specification

This chapter contains the test cases used in the system test and in which order the tests will be executed.

10.1 Purpose

This test is taken place to ensure that the product behaves as the group expects it to behave. It is also meant to check the user performance of the product. The test cases are written to make it easier for the test constructors to create the tests.

10.2 Description

Each test will be stated as below with a test id as ST-X where X increments with the tests.

Test id	Identification of the acceptance test.
Purpose	Describes why the test is being done.
Req.	The corresponding requirement
Req. level	The level of the corresponding requirement.
Data input	The data that has to be inserted into the system to be able to carry out the test.
Execution	Describes how the test should be carried out.
Expected result	If the test result is the same as the expected result, the test has passed.

Test scripts and surrounding code will be created during the implementation phase.

10.3 Test order

This test phase will start with the test ST-1 and then continue with increasing order.

10.4 Account for satisfactory system test

During all tests the time each execution takes will be taken into consideration. This is done to ensure that the performance of the system is satisfactory.

Since we do not know what classes that needs to be updated or created to reach the Normal and Extra requirements yet, they are not listed in the system testing. These tests will be added as the project progresses.

10.5 Test cases.

Test id	ST-1
Purpose	To verify that the program is written in C++ and Flex.
Req.	N-1
Req. level	Basic
Data input	-
Execution	Go through the code and examine if it is written in C++ and Flex.
Expected result	The code is written in C++ and Flex.

Test id	ST-2
Purpose	To verify that the program is documented in Doxygen.
Req.	N-2
Req. level	Basic
Data input	-
Execution	Go through the comments and examine if it is commented with Doxygen comments.
Expected result	The code is commented with Doxygen.

Test id	ST-3
Purpose	To verify that the program is easy to upgrade in the future.
Req.	N-3
Req. level	Basic
Data input	-
Execution	Control the written code towards the Programming handbook [Lissing, 2005:1].
Expected result	The customer thinks that the program is easy to upgrade

Test id	ST-4
Purpose	To verify that the program's code and user interface as well as the documentation are all written in English.
Req.	N-5
Req. level	Basic

Data input	-
Execution	Go through the code and documents and see if they are in English.
Expected result	The code and documentation are written in English.

Test id	ST-5
Purpose	To verify that the program's code is handled with Subversion.
Req.	N-6
Req. level	Basic
Data input	-
Execution	Check the Subversion history for existing versions.
Expected result	The Subversion history contains several versions of code.

Test id	ST-6
Purpose	To verify that the program has the ability to parse Verilog code.
Req.	F-1
Req. level	Basic
Data input	Verilog file.
Execution	Execute the program.
Expected result	The program executes without errors.

Test id	ST-7
Purpose	To verify that the program can produce a syntax tree from a Verilog code file.
Req.	F-2
Req. level	Basic
Data input	Verilog file.
Execution	Execute the program.
Expected result	The program produces a syntax tree that contain nodes for all constructs in the Verilog source file that are important to the generation of documentation.

Test id	ST-8
Purpose	To verify that the program prints the syntax tree in a readable form.
Req.	F-3
Req. level	Basic
Data input	Verilog file.
Execution	Execute the program.
Expected result	The program prints a readable syntax tree that contain nodes for all constructs in the Verilog source file that are important to the generation of documentation.

11 Acceptance test specification

This chapter contains a reference to the test cases for the acceptance test and the order for them to be executed.

11.1 Purpose

This test is conducted to ensure the customer and the project group that the product has met all the expected requirements which the customer and the group have agreed upon. The test cases are written to make it easier for the test constructors to create the tests.

11.2 Description

Each test will be stated as below with a test id as X where X increments with the tests.

Test id	Identification of the acceptance test.
Purpose	Describes why the test is being done.
Req.	The corresponding requirement
Req. level	The level of the corresponding requirement.
Data input	The data that has to be inserted into the system to be able to carry out the test.
Execution	Describes how the test should be carried out.
Expected result	If the test result is the same as the expected result, the test has passed.

Test scripts and surrounding code will be created during the implementation phase.

11.3 Test order

The testing will start with test ’1’ and then continue in increasing order.

11.4 Account for satisfactory system test

All the active requirements are tested in the acceptance testing.

11.5 Test cases

All the acceptance test cases are listed in Requirements specification [Hilding, 2005].

12 References

12.1 Internal documents

- [Hilding, 2005] Hilding, Daniel, "Requirements specification" (2005)
- [Jormedal, 2005] Jormedal, Martin, "Project plan" (2005)
- [Norling, 2005] Norling, Jonas, "Architecture specification" (2005)
- [Lissing, 2005:1] Lissing, Johan, "Programming handbook" (2005)
- [Lissing, 2005:2] Lissing, Johan, "Design specification" (2005)
- [Åberg, 2005] Åberg, Eric, "Test report" (2005)

12.2 External documents

[Arvedahl, 2002] Arvedahl, Svante, "RUT - Development Handbook 12.1 Choice of Test Strategy v6.0" (2002)

[Karlsson, 2005] Karlsson, David, "RUT - development tutorial 14.1 Acceptance test v 2.0" (2005)