D37.1 Requirements Report

Client Testbed

Document Identification	
Date	02/05/2013
Status	Final
Version	Version 1.0

Related SP / WP	SP3 / WP37	**Document Reference**	https://dms-prext.fraunhofer.de/livelink/livelink.exe/overview/2968975
Related Deliverable(s)	D22.x, D31.1, D32.2, D33.1, D34.1, D35.1, D36.1	**Dissemination Level**	PU
Lead Participant	USTUTT	**Lead Author**	Eray Özmü
Contributors	Christopher Ruff, Urmo Keskel, Dr. Lothar Firtsch, Jens Kubieziel	**Reviewers**	IBM, UNEW

This document is issued within the frame and for the purpose of the FutureID project. This project has received funding from the European Unions Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 318424

Document Information

Contributors

Name	Partner
Eray Özmü	USTUTT
Christopher Ruff	USTUTT
Urmo Keskel	SK
Dr. Lothar Fritsch	NRS
Jens Kubieziel	AGETO

History

Version	Date	Author	Changes
0.1	22.04.2013	Eray Özmü, Christopher Ruff, Urmo Keskel, Dr. Lothar Fritsch, André Gutwirth, Jens Kubieziel	Integration of Wiki-Content into Word-Format
0.2	22.04.2013	Eray Özmü	Outline refined
0.3	23.04.2013	Lothar Fritsch	Chapter Security
0.4	24.04.2013	Christopher Ruff	Added Chapter Test Terminology
0.5	25.04.2013	Eray Özmü	Refined chapters and added content
1.0	02.05.2013	Eray Özmü, Christopher Ruff, Lothar Fritsch, Jens Kubieziel	Corrected and improved document based on the recieved reviews.

Table of Contents

Abstract

For projects like FutureID with a large and distributed development team, testing is an important factor of success. Errors in software often lead to frustrated users and a low acceptance rate of the software itself.

Therefore it is intended that the FutureID-Client will be extensively tested. Extensive testing is not possible if the testing activities are not supported by the right toolset, as manual tests are time consuming and expensive.

This leads to the importance of automated tests. In order to keep the rate of testing high, the FutureID-Client will be tested with automation tools. Unit tests and integration tests must be executed automatically, whereas system tests and acceptance tests should be tested automatically, if possible. The results of these tests should then be reported to the designated developers and presented on a central platform.

Further important parts of software testing are documentation, separation of responsibilities, aspects to test (security, usability, crash tests and functional tests) and the platforms and tokens which need to be tested. As FutureID aims at supporting all different kinds of tokens, the testbed should support testing of as many tokens as possible. A set of a few cards however must be supported by the test environment for sure.

1. Introduction

The FutureID project aims to build a multi-platform eID client which is secure, privacy-friendly and usable as well. These multi-disciplinary requirements and the fact that the development team is geographically separate lead to a complex software development, which is prone to have bugs or security issues.

Therefore it is even more important to be sure that the development team is supported by the right testing tools and testing activities. In order to ensure that the testing activities meet the requirements of the developers and the FutureID-Client, this document will look into different aspects of testing and the requirements of the FutureID-Client, like complexity, user workflow etc. This document will then provide requirements for the testing activities of the FutureID-Client, which will be transformed in Task 37.2 to a test strategy.

As testing has evolved to a major part of software development, tools have been developed to automate and support testing activities. The purpose of the client-testbed is to provide the right tool set and the right platform, to be able to test the software early and thoroughly to save costs on the long run.

The methodology for testing will be based on the W-Model which describes a parallel and early testing during the whole project. It also includes incremental development and testing.

This document is mainly intended for the task 37.2, which will create a testing strategy based on these requirements and the special requirements of the FutureID-Client. The ideas and requirements will be specified and evaluated more deeply within this task.

2. General Requirements

In this section the general requirements for the client testbed will be defined. As defined in Deliverable D23.2, there are different development models.

2.1 W Model

The view of testing has changed over time. In the 1950's testing had the character of a mere demonstration and should show how software worked. 20 years later software engineers used tests to find defects in their software, while in the 1980's testing started to have a predictive character. It should measure the quality of software. Software testing in current times is used to control the quality of the underlying software and to prevent defects.

Rudolf van Megen, founder and CEO of SQS Software Quality Systems AG, evaluated about 3.000 IT projects [1]. He found out that the cost to fix software defects rises in a near exponential manner. From the requirement phase to specification of software the cost quadruples. The factor stays the same from specification to implementation to acceptance. When the software is used in the field the trouble-shooting costs are ninety times higher compared to the requirement phase. Those numbers are also supported by other studies.

The process of software development followed the so-called waterfall model for a long time. This model separates the process in different phases and only if one phase is completed one can move to the next [2]. Testing itself was a step which was done in a late phase. Thus defects were uncovered late and the costs to fix them were quite high. This was one of the reasons why 17 software developers published a Manifesto for Agile Software Development in 2001 [3]. The idea of the agile approach is to have an iterative and incremental development. It breaks tasks into small pieces which require little or no planning. With this approach developers can react to change requests quite fast and it is even no problem to change the requirements in late phases of software development. Testing is done throughout the whole process.

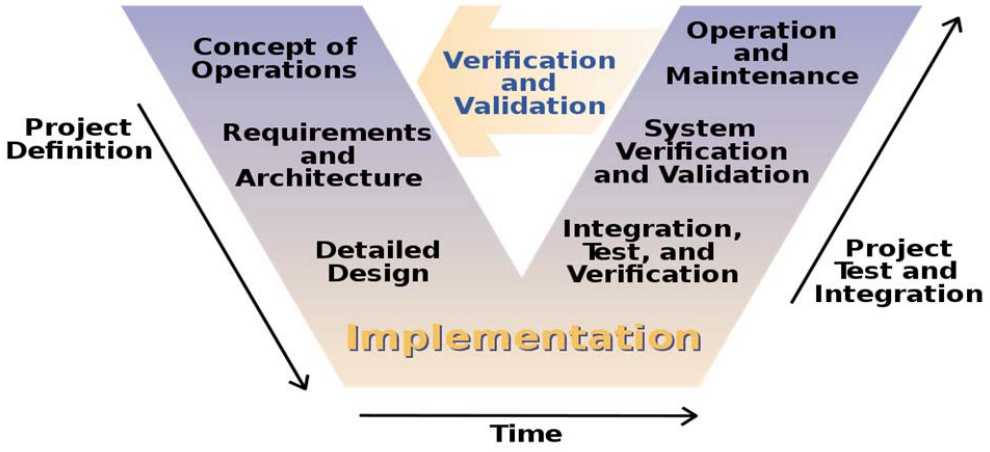

Figure 1 - The V model (By Leon Osborne, Jeffrey Brummond, Robert Hart, Mohsen (Moe) Zarean Ph.D., P.E, Steven Conger ; Redrawn by User:Slashme. [Public domain], via Wikimedia Commons)

Starting from the waterfall model the V model (Figure 2) was developed [4]. The V model starts also at the conceptual point and moves to the coding phase. But after that the steps go upwards and build the typical V shape. The upward steps are typical test and verification tasks which are connected to the left side. However there were several critics of that model and this led to developing an extension, the so-called W model, was developed.

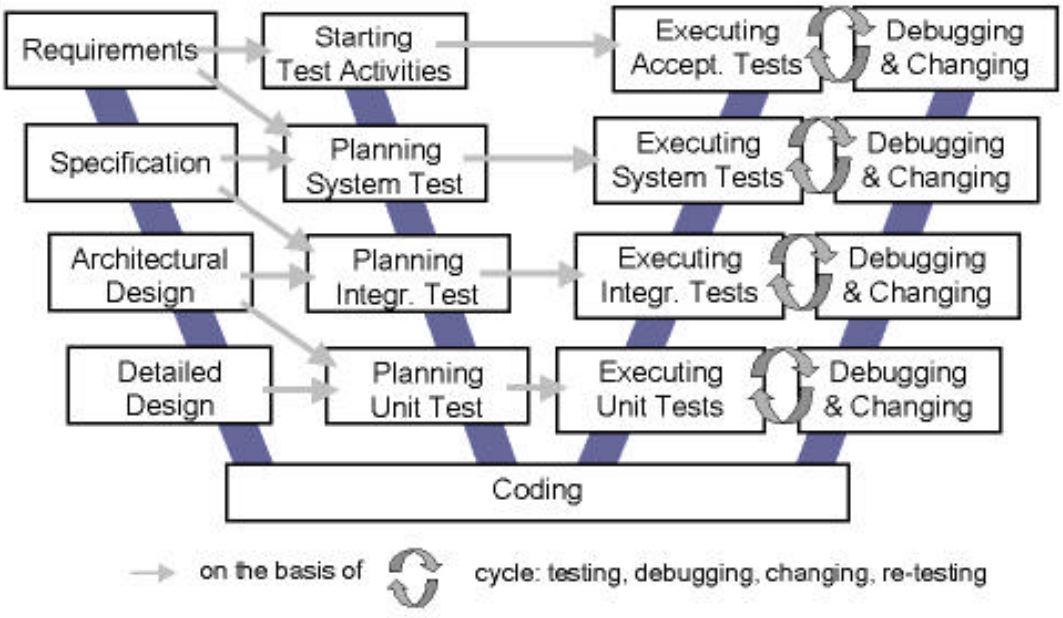

Figure 2 - W-Model Derived from [5]

The W model was developed in 1983 and is used in Agile development today. Starting from the V model each step gets a neighbour. This neighbour is a test activity. So the first steps are writing requirements, specification and design of a system. Each of those steps has a test step on its side. When it comes to writing source code the neighbour step is unit testing. Unit test are small test which examine only a small part of the software. Paragraph 3.1.1 explains it in more detail. The upward steps from the V-Model, build the software, the system and then install the system, are accompanied by integration, system and acceptance tests. While the first, integration tests, examine larger software components, system tests deal with the whole computer system usually consisting of hard- and software (see paragraphs in 3.1). So basically the model has two “V”s besides each other which creates the typical W shape [2].

Before it comes to programming all test cases are specified and more than half of all test activity is finished. It simplifies the test infrastructure and usually defects are recognised in an early phase. This helps to keep the cost of fixing defects low.

As it was shown above the testing process should start quite early in the development process. This ensures that bugs or defects are found in early stages. Following the W model seems like the best alternative.

Furthermore we will need small, manageable test cases so that they are quick to execute and probably can be automated. Testers or test managers should have an eye on exact test coverage to get the required functionality. As pictured before there is a wide range of test participants. The following paragraphs will concentrate on developer, tester and test coordinators.

A basic requirement for a test case is a description of the test together with test data. It should describe the functionality and the expected results. Commonly a matrix is used. Within that matrix each row represents a test case and a column represents a different scenario and the expected results. Afterwards one has to identify the test conditions of a test case. Usually several conditions must be met to execute a test case. Those conditions must be identified and sorted into the three categories (V) valid, (I) invalid and (N/A) not applicable. In the last step it must be made sure that the requirements for the use case are met. So it should include parameters input data, performance numbers and used protocols. These values are entered into the matrix and if done the test case can be executed.

Once test data are planned one can apply test tools. They help and support the testing process in a wide range of tasks. It starts with the determination of test cases and the creation of test data. Furthermore those tools help to execute and manage tests. In the end they also create very detailed tests reports. Please see Deliverable 37.2 Test Strategy for further details.

These detailed test reports will later on be used to evaluate the quality of the development of different modules and the overall system. Metrics should be designed in the later works for the client testbed.

2.2 Scope

A scope has to be defined as the potential test cases are limitless and have to be reduced to a reasonable set. We should approach this using a requirement-based approach in combination with a risk-based approach. The test cases should cover as much of the required functionality as possible. As the project involves a lot of security aspects the risk-based approach should also be employed prioritizing testing functionality that has a high cost of failure.

2.3 Test according to established standards

Standards describe terms, concepts, techniques and procedures agreed upon by different parties. While conforming to certain standards requires more effort and expenses by the implementing party, there are many benefits. The compliance with standards in testing improves the overall understanding of the process and the vocabularies used, as most developers and testers might already have got in contact with these standards in the past. The use of standards is a key to ensuring quality control, having clearly formulated requirements and specifications to test against. Manufacturers also benefit from using standards as they often rely on previous experience, know-how and best practices developed over multiple iterations, avoiding known pitfalls, resulting in a higher quality end product.

While the compliance with standards is commonplace and absolutely necessary in many industry fields, these standards are often not fully adopted in software engineering which might be due to various reasons such as the industry being relatively young and the used technology and tools constantly evolving, resulting in changes of the requirements and standards. Still, a comprehensive set of software standards exists today which should be complied with where possible and sensible. In this deliverable we concentrate on the subset of software engineering standards considering software testing. We identified several standards that are relevant to software testing. We encourage the compliance to these standards by the implementing partners in the Future ID project.

Standard	Purpose	Comment
BS 7925-1	Testing Vocabulary	British Standard, established test terminology
ISTQB Glossary of Testing Terms	Testing Terminology	Rather new, not as widely adopted terminology
BS 7925-2[4]	Software Component Testing Standard	British standard, comprehensive guideline for

		software testing
IEEE 829	IEEE Standard for Software and System Test Documentation	Comprehensive international standard with useful templates for documentation

Table 1 - Relevant Testing Standards

2.4 Test Terminology

Defining and using a common terminology is important for efficient communication between all involved stakeholders. The terminology that is agreed on should be functional and fit the requirements of the given context. In the domain of software testing, there are several standards that can be employed. While the ISQB standard for testing terminology is newer than BS 7925-1 [6] we suggest to comply with BS 7925-1, since it is an established standard that other proposed testing standards are based or at least reference it.

2.5 Component Testing

The component testing and corresponding test process should be modeled around the BS 7925-2 standard. It comprehensively describes a structured test process with all necessary stages for planning, specification, execution, recording and completion of tests as well as techniques and best practices for the design of test cases and measurement of test completion.

2.6 Test Documentation

An appropriate documentation of the test process ensures that the process is traceable and reproducible. The IEEE Standard for Software and System Test Documentation (IEEE 829 [7]) provides a guideline for producing different documentation elements depending on the testing and integrity levels defined for each phase. The documents specified in IEEE 829 are listed in table 2.

Table 2 – Test Documentation

Document	Purpose
Master Test Plan (MTP)	Describes overall test planning, management, resources, responsibilites and activity
Level Test Plan (LTP)	Describes the scope, approach, resources, schedule for a specific test level
Level Test Design (LTD)	Refines the test apporach,lists the features tob e tested with pass/fail criteria
Level Test Case (LTC)	Defines input/output information for testcases of the associated LTD

Level Test Procedure (LTPr)	Describes the steps for the execution of test cases
Level Test Log (LTL)	A chronological record of details regarding the execution of tests
Anomaly Report (AR)	Documents incidents or events during testing that require investigation
Level Interim Test Status Report (LITSR)	Summarizes the results of the testing, provide recommendations, during test activities
Level Test Report (LTR)	Summarizes the results of the testing, provide recommendations (one for each test level, but can be merged)
Master Test Report (MTR)	Corresponds to the MTP, summarizes test results, provides recommendations

Not all of these documents have to be produced. It is advisable to look at the importance determined by the cost of failure of certain parts of the software or development cycle to identify the most relevant documentation. Also documents don't have to be stand-alone but can be combined to avoid overlap and duplicated efforts. The overall goal is to have an adequate set of test documentation. We suggest the compliance with IEEE829, as proposed in Deliverable 23.3.

R 2.3.1: The testing vocabulary should be based on BS 7925-1 as much as possible.

R 2.3.2: The testing documentation should be based on IEEE 829.

R 2.3.3: The component testing should conform to BS 7925-2.

2.7 Test Automation

Test automation is a crucial part of software testing. As software gets more and more complex, testers have to find a way to catch up with the complexity of the tested projects. The FutureID-Client itself is also a very complex project with many interfaces and a large development team. For this project it is highly recommended to use test automation tools that automate different tasks in testing.

Different tasks of the testing lifecycle can be automated with the right toolset. This does also include highly time consuming tasks like system tests. The most advanced way of test automation is the automation of the whole test process which can be triggered by different events, like code commit to the code repository.

As [8] states, it is important to start with test automation at one point. Otherwise the testing team will be too busy with manual tests to build a toolset with automated tests. Figure 3 can be seen as a motivation for the testing team to start with test automation right at the beginning of the project.

Figure 3 - Vicious circle of test automation [8]

As FutureID is a project that just started, it is important to not get into this vicious circle and use test automation as a starting point for testing. But there are also hurdles to overcome in this project regarding automation of testing. The flow of user interaction includes browser software and the FutureID-Client as well. Normal user interface automation is not sufficient to automate the whole user interaction for a specific task. Therefore automation tools must be found, which use image recognition to automate real user interaction in a black-box manner.

R 2.4.1: The testbed must provide tools to automate the testing process, or at least different testing tasks. This must include tools to automate user interaction based on image recognition.

3. Test Levels

3.1 Definition of Test Levels

During the software development process there are different kinds of tests. The Software Engineering Body of Knowledge (SWEBOK) has defined three testing levels in their 2004 edition and the current reviewed) SWEBOK V3 (to be released in 2013adds some more. The following paragraphs will explain the different levels.

3.1.1 Unit testing

Unit tests take place on the source code level. A software developer verifies the functionality of a subprogram or specific code parts. Besides from the actual working code the developer writes a test case which ensures the correct working of its code. A unit test case gives several valid and invalid inputs and defines the behaviour of the code or subprogram. If it gives the correct expected output the code is considered to be correct.

Because of their nature unit tests find problems in source code quite early in the development process. A very popular approach to software development derives from the Manifesto for Agile Software Development. Methods, which are included here, are for instance eXtreme Programming (XP) and Scrum. Both methods make large-scale use of test-driven development (TDD). In this process the developer writes the test case first and produces code which passes the test afterwards. Those tests are usually unit tests.

While TDD programmers write more code the total implementation time is shorter than in traditional models. Furthermore test cases help to find defects in the source code.

3.1.2 Integration testing

After the unit tests comes integration testing. During the process different software components are packed together and tested as a group. So integration tests verify the interaction between software components. Contrary to unit testing there is no source code needed. Integration tests often happen as so-called black box tests, where the tester has no insight to the inner workings of the software.

With the Big Bang approach the software is put together and forms a complete system (or at least a major part of it). This system is then used for testing.

In the top-down case the tester starts with a high-level system and tests it down to subprograms. If no subprograms exist, stubs are created.

Bottom-up testing starts from subprograms or small components and integrates them step-by-step until the complete system is tested.

There is a fourth strategy called sandwich testing. This combines bottom-up and top-down testing. In practice this is more often used than any of the above mentioned approaches alone.

In Deliverable 37.2 the approach for integration testing will be evaluated, however we recommend to use the bottom-up approach to avoid long and unpredictable testing phases by late integration.

3.1.3 System testing

System testing looks at the system as a whole. It simulates the future software environment and is filled with real-world test data. At this level of testing a comparison of interfaces to other applications, to hardware or to the operating system is done. The system is compared to non-functional requirements.

An important part of system tests are installation tests. The software is installed on the target system and will be verified if it passes the hardware requirements. However installation testing is usually done after acceptance testing.

3.1.4 Acceptance testing

Finally acceptance tests should find out if the initial requirements of the contractor are met. Therefore the tests are run on the target system. So acceptance tests are the final phase of software development.

3.2 Separation of responsibilities

As defined in the Description of Work, there will be a set of test tools and the test environment provided for the according tasks. The partners involved in the testing tasks of each work package are responsible for running the specified tests in the specified time frames. Furthermore the different test levels will be assigned to different roles in the project.

R 3.2.1: Unit tests must be performed by the developers themselves. This also affects the rules of checking code into the repository. All unit tests have to be passed, before checking any code into the repository. There may also be a second code reviewer before the check-in.

R 3.2.2: Integration tests must be performed on a specified time basis.

R 3.2.3: Integration tests should be performed automatically by the test environment right after every build of the client.

R 3.2.4: The results of the integration tests must be presented on a central platform and be visible for every developer.

R 3.2.5: Critical bugs discovered by integration tests must be reported instantly to the assigned developers, or the work package leaders.

R 3.2.6: System tests should be performed automatically. This level of testing should be performed on the specific platforms, which will be defined later on. The aim of this test level is to check the functionality in real running conditions. As there is third party hardware involved in this type of testing, the whole setup will be tested - also including the FutureID server.

R 3.2.7: Acceptance tests should be performed automatically by setting measures for general acceptance criteria.

4. Aspects to test

All software products can be divided in different aspects that can be tested separately. In order to understand the special requirements for each aspect, a list of the relevant aspects for the FutureID-Client will be created and rated. Upon the different aspects, specific tools can be identified and used to meet special requirements for each aspect.

In order to overcome special problems which are likely to appear in FutureID by the geographically distributed development team and the multi-disciplinarity within the team, a central testing environment must be created for the testbed.

4.1 Security

The client development will provide frequent updates of client prototypes. The client will most likely incorporate open source components and other software like device drivers not designed by FutureID. The tests are focused on quality control, and have to be performed quick, with little resource use, and widely automated. The testing design will try to install metrics for cues on the presence of security mechanisms as far as they are measurable.

We focus on three main areas of security-requirements:

1. Client security configuration management;
2. Client security protocol compliance;
3. Client user interaction in critical security actions.

Normal testing with parameters, intervals, function calls and parameter errors (to provoke buffer overflows etc.) will be performed with an automated testing tool.

The following sections define and describe the three areas in more detail.

4.1.1 Client security configuration management

When using the client, we need to ensure correct versions and patch levels of all platforms, components and apps/applets. Since operating systems and drivers for e.g. card readers and smart cards constantly get updated, we need a specification of versions and components that constitute the “secure” and “approved” environment for the FutureID client. From this list, we should ensure that:

- The used platforms and components are installed on the test environment in the required versions;

- That we have control over when potentially automatic updates of components on the base platforms happen such that we can coordinate with the developers concerning possible change needs in the FutureID client;
- That we have a mechanism for version control, patch control and version/requirements synchronization between the development requirements and the actual client testbed.
- By injection, manipulation and replay of messages, we will be able to test exception handling in critical protocols.

In consequence, we need a description about software versions, patch levels, and other relevant execution environment components that we can detect and compare, best with an automatic mechanism that will be run whenever the client software gets updated.

4.1.2 Client security protocol compliance

The FutureID client will execute many security protocols that will be used for establishing identities, verifying certificates, checking validity of the certificates, and for the generation of signatures. Since parts of the functionality are moved to the server side, we need an understanding for how these security functions are distributed between the client and the server. For ongoing software testing, it is advisable to use protocol sniffers and communication monitoring software on the client testbed platform in a similar way it was used in [9] that can help us detect that the client:

- Actually connects to and communicates with the FutureID server, the CAs, the SmartCard, or any other necessary entity outside the client.
- The observations of message flow could constitute “patterns” that can tell us whether there can possibly happen a server or a client authentication, a certificate validation or other actions, even when the messages flows are inside TLS/SSL connections.
- By checking the specification against the observed message flow patterns, we will be able to detect missing interactions (e.g. acceptance of authentications or certificates without checking with directory services). Some tools for protocol analysis will help achieving this task.

To enable the implementation of the above suggestions, both the specification and the infrastructure in FutureID have to prepare for intervention from the test team. On the client side, it should be possible to log file message exchanges, e.g. by inserting log level parameters in compile time and runtime code. The logs could, depending on log levels, provide information on message exchanges, used cryptographic functions, and their sequence.

4.1.3 Client user interaction in critical security actions

User's assessment of the security and trust situation is an important factor when running the client. Many signals must be processed – queries for passwords, validity of SSL/TLS certificates, and the validity of the execution platform are some of them. Exception handling, such as processing error messages, handling exception states (invalid PIN entry, unreachable CA directory service) must be communicated to the user. The user then should receive a clear description of the problems and intstructions on their mitigation (which can be a challenge in large, multi-stakeholder distributed systems). We suggest to compile a test list list when the specifications and requirements are available. They should include:

- PIN entry
- Token interactions
- Exception states (server, CA, application not reachable; card removed under operation, card locked, card blocked, card or certificates don't qualify for service, unexpected messages, protocol failures)
- User interface visualization of exception states
- User intervention possibilities in case of exception states
- Visualization of trust information
- Transaction history browsing (audit trail) for previous transactions, including a log over provided personal data. This is important for privacy audits and user interventions.

To test the above topics, it may be necessary to be able to provoke exception states, e.g. blocked PIN codes, unreachable CAs and protocol failures between client and server. Specification of client development should accommodate for possibilities to provoke such exceptions

4.2 Usability/Acceptance Tests

Usability is a factor that is essential for the long term success of the FutureID-Client. The higher the acceptance for the client is, the more it will be used and preferred against other products. Acceptance of software is defined by different measures and can partly even be tested automatically.

Acceptance tests inspect whether the FutureID-Client is able to deliver the functionalities that the users ask for. These tests normally are tested manually by the users themselves, which may happen only rarely [10].

R 4.2.1: The testbed should provide tools for acceptance tests.

4.3 Crash Tests

With crash tests, the testers try to cause crashes in the software. As functional tests are not suitable to detect all bugs in an application, this additional method of testing has been introduced. One possible method to cause crashes to the software is to provoke states, the developers and testers didn't think of. This can be done by creating random input.

A well-known method for random input tests is the so called "monkey test". This type of tests simulates random user interaction with a certain amount of touches/interactions and tries to crash the software. If the certain amount of interactions was done without crashing, the test is passed.

R 4.3.1: The testbed should provide a way to create random input tests to provoke states that were no anticipated.

4.4 Functional Tests

Functions are the most common aspects for average testers. Most projects do manual functional testing. In these cases the functions get listed and will be tested with system tests on a certain basis.

For a large project like FutureID with many intersections and an interdisciplinary team, it is recommended to automate functional tests as far as practicable.

R 4.4.1: The testbed should provide as many functional tests automatically as possible.

5. Testing Targets

5.1 Target Platforms

The Client-Testbed includes testing on the following platforms:

5.1.1 Operating Systems

Windows 7, 32 bit

Browsers: Internet Explorer, Mozilla Firefox, Chrome

Windows 8, 64 bit

Browsers: Internet Explorer (including Metro version), Mozilla Firefox, Chrome

Ubuntu Linux, 64 bit

Browsers: Firefox

Macintosh 10.8

Browsers: Safari, Firefox

Android 4.1

Browsers: Chrome

All operating systems are tested with last available patch level. The patching of systems will be carried out at least once per month. Testing browsers will include only latest stable versions.

5.1.2 Java Runtimes

OpenJDK 6

OpenJDK 7

OracleJDK 6 x64

OracleJDK 7 x64

R 5.1.1: The FutureID-Client should be tested with the operating systems and browsers mentioned in chapter 5.1.

5.2 Target Tokens

Target Tokens are an important part of the testing of the FutureID-Client, as FutureID aims at the use of all different kinds of cards.

Automatic testing with client testbed should include following tokens, derived from Deliverable D32.1:

SmartCards:

Austria

Belgium

Czech Republic

Finland

Estonian ID-card (3.4 and 3.5 versions)

German eID card

German health card

Italy

Liechtenstein

Lithunia

Monaco

Portugal

Spain

Switzerland

R 5.2.1: The client testbed must support at least the Estonian ID-card and the German eID card for system tests.

6. Conclusion

Testing is an important part of the whole development lifecycle. In order to create maintainable software that meets the specified requirements, early testing is needed. This is why we focussed on the W-Model and aim to an early, automated and central testing.

The essence of this document is the list of all requirements:

R 2.3.1:	The testing vocabulary should be based on BS 7925-1 as much as possible.	SHOULD
R 2.3.2:	The testing documentation should be based on IEEE 829.	SHOULD
R 2.3.3:	The component testing should conform to BS 7925-2.	SHOULD
R 2.4.1:	The testbed must provide tools to automate the testing process, or at least different testing tasks.	MUST
R 3.2.1:	Unit tests must be performed by the developers themselves. This also affects the rules of checking code into the repository. All unit tests have to be passed, before checking any code into the repository.	MUST
R 3.2.2:	Integration tests must be performed on a specified time basis.	MUST
R 3.2.3:	Integration tests should be performed automatically by the test environment right after every build of the client.	SHOULD
R 3.2.4:	The results of the integration tests must be presented on a central platform and be visible for every developer.	MUST
R 3.2.5:	Critical bugs discovered by integration tests must be reported instantly to the assigned developers, or the work package leaders.	MUST
R 3.2.6:	System tests should be performed automatically. This level of testing	SHOULD

should be performed on the specific platforms, which will be defined later on.

R 3.2.7:	Acceptance tests should be performed automatically by setting measures for general acceptance criteria.	SHOULD
R 4.2.1:	The testbed should provide tools for acceptance tests.	SHOULD
R 4.3.1:	The testbed should provide a way to create random input tests to provoke states that were no anticipated.	SHOULD
R 4.4.1:	The testbed should provide as many functional tests automatically as possible.	SHOULD
R 5.1.1:	The FutureID-Client should be tested with in chapter 5.1 mentioned operating systems and browsers.	SHOULD
R 5.2.1:	The client testbed must support at least the Estonian ID-card and the German eID card for system tests.	MUST

7. References

[1] R. van Megen, “Early Test preparation,” in *Supporting Customer - Supplier Relationship: Requirements Engineering and Quality Assurance*.

[2] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, *Manifesto for Agile Software Development*. 2001.

[3] D. Graham, *Foundations of software testing: ISTQB certification*. Australia: Course Technology Cengage Learning, 2008.

[4] A. Spillner and H. Bremenn, “The W-Model Strengthening the Bond Between Development and Test,” in *Proceeding of the STAReastt’2002 Conference, Orlando, Florida*, 2002.

[5] S. Desikan and G. Ramesh, *Software Testing: Principles and Practices*. Dorling Kindersley/Pearson Education, 2006.

[6] British Standards Institution, *Software testing. Part 1. Part 1, Vocabulary.* [S.l.]: B S I, 1998.

[7] “IEEE standard for software test documentation,” 1998.

[8] K. Kaczor, “How to Start with Test Automation?,” *Agile Testing*. 23-Dec-2010.

[9] L. Fritsch, A.-K. Groven, and L. Strand, “A Holistic Approach to Open-Source VoIP Security: Preliminary Results from the EUX2010sec Project,” *International Conference on Networking*, vol. 0, pp. 275–280, 2009.

[10] M. Finsterwalder, “Automating acceptance tests for GUI applications in an extreme programming environment,” *XP2001*, 2001.