D37.2 - Test Strategy

Client Testbed

Document Identification	
Date	16/05/2013
Status	Final
Version	1.0

Related SP / WP	SP3 / WP37	**Document Reference**	https://dms-prext.fraunhofer.de/livelink/livelink.exe/properties/3002288
Related Deliverable(s)	D37.1	**Dissemination Level**	PU
Lead Participant	USTUTT	**Lead Author**	Eray Özmü
Contributors	Christopher Ruff, Jens Kubieziel, Urmo Keskel	**Reviewers**	FHG, TUG

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

Abstract

The *FutureID* client is a complex software product with strong requirements regarding security and usability. In order to make sure, that the software quality is sufficiently high, it is essential to develop a test strategy and provide the development teams with the right tools.

The provided tools will be of a general character. It is not intended to provide specialized tools, for example, for protocol testing. However a sufficient test toolset will be provided to enable automatic testing at different levels. Unit tests and integration tests will be implemented by using TestNG[1]. System tests will be supported by Sikuli[2] for computers and Robotium[3] for Android devices.

All information and tools will be brought together within the central continuous integration platform Jenkins[4]. This will lead to highly automated testing and detailed test reports on a regular basis.

[1] http://testng.org/doc/index.html
[2] http://www.sikuli.org/
[3] https://code.google.com/p/robotium/
[4] http://jenkins-ci.org/

Document Information

Contributors

Name	Partner
Christopher Ruff	USTUTT
Eray Özmü	USTUTT
Jens Kubieziel	AG
Urmo Keskel	SK

History

Version	Date	Author	Changes
0.1	28.01.2013	Eray Özmü	Document setup Structure of document
0.2	29.01.2013	Eray Özmü	Added content to chapters: Introduction, Testing Approach
0.4	02.05.2013	Jens Kubieziel	Added content to chapter System Testing
0.7	04.05.2013	Eray Özmü	Added content to remaining chapters
1.0	16.05.2013	Eray Özmü, Christopher Ruff, Jens Kubieziel, Urmo Keskel	Finalized

Table of Contents

Abstract 1

Document Information 2
Contributors 2
History 2

Table of Contents 3

1. Introduction 5

2. Scope and Limitations 6
2.1 Scope 6
2.2 Limitations 6

3. Testing approach 7

4. Test levels 8
4.1 Unit tests 8
4.2 Integration tests 8
4.3 System tests 9
4.4 Acceptance tests 9

5. Test tools 10
5.1 Central continuous integration platform 11
5.2 Unit tests 12
5.3 Integration tests 13
5.4 System tests 13
5.4.1 Relevant platforms 14
5.4.2 Windows, Linux and Macintosh 15
5.4.3 Android 17
5.5 Acceptance tests 19

6. Test documentation 20
6.1 Test cases 20
6.2 Test results 21
6.2.1 Build success 21
6.2.2 Code coverage 22

7. Success factors 23

8. Conclusion 24

9. Bibliography 25

1. Introduction

FutureID is a complex project with 19 partners from 11 European countries and, thus, a big development team. The architecture of the software is highly modular and the variety of supported devices is large. This leads to certain requirements for the testing environment of the *FutureID*-Client, which have already been stated in the Deliverable D37.1. This document uses these requirements and renders a testing strategy for the *FutureID*-Client, which should be read by all partners who are concerned with the development or testing of client components.

It is highly recommended to read through the testing strategy and act accordingly, to ensure a high quality of the developed client. Only by testing the client early and thoroughly, the client will be able to meet the high security and quality requirements.

This document will furthermore help to organize testing activities during the project and determine the entry and exit criteria of testing, testing suspension and resumption criteria. This document also describes types of tests, which will be used in this project and the test tools, which will be used to support testing activities.

2. Scope and Limitations

Within the *FutureID* client, there are modules that are concerned with the smartcard interfaces, cryptographic modules, and user interfaces of the client. As these disciplines are completely different and testing for each discipline requires strong knowledge in the respective discipline, we will only provide a general view on the testing tools.

2.1 Scope

The scope of the client testbed is to provide tools for testing in order to ensure a high quality and secure software. The provided toolset and documents will cover different levels of testing.

Especially the most common platforms will be part of the system tests. The unit tests of the different modules as well as the integration tests are platform independent and will be executed with the help of special tools.

2.2 Limitations

It is not intended to test all aspects of the software automatically, as this goal would not be feasible. Furthermore, it is not intended to provide special tools for tests on, for example, a protocol level or other too detailed aspects. At this point we rely on the help of the developers of these specific aspects of the software.

Due to the large number of platforms, we are not able to perform system tests on all of these, but will limit the system tests to a set of commonly used platforms in their current versions.

3. Testing approach

Different approaches have emerged in the previous years in order to structure testing activities systematically. Deliverable D37.1 provides an in-depth view on the different approaches and recommends the usage of the bottom-up approach, which fits best on the *FutureID*-Client.

In order to make sure, that all partners have a common understanding on how the testing activities are coordinated within the project, this approach will be explained in detail within this chapter. We recommend all partners to follow these rules during development.

The setup of different levels of testing leads to the following automated testing process, which can be seen in Figure 1. The code first needs to be pulled and built. Already at this stage bugs are possible to appear.

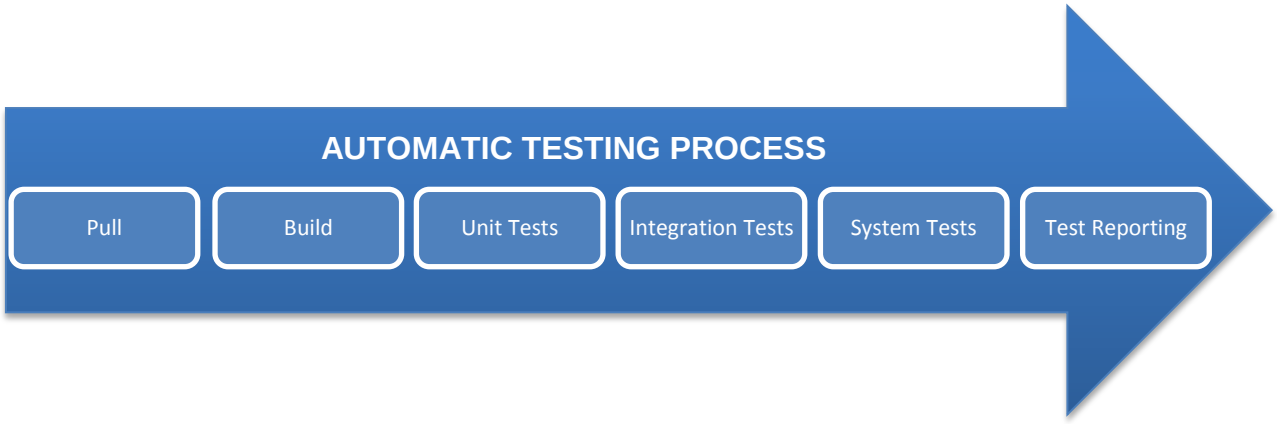


Figure 1 - Testing process

If the build was successful, the next stage of testing will be run. The unit tests will be executed and analysed. These are the unit tests the developers wrote for their developed methods and classes. If the unit tests passed successfully, the integration tests will be executed.

If a major bug is found, the tests won't be executed further to save resources and time.

4. Test levels

The task 37.1 already dealt with the different types of tests and the roles and responsibilities of the test levels and test types. The essence of the requirements regarding test levels is the following: There are four main test levels. At the lowest level are unit tests, followed by integration tests, system tests and acceptance tests.

We will provide test tools for all of the mentioned test levels but need the responsible roles and partners to follow their responsibilities of testing.

Test Level	Responsible Role	Tools
Unit Tests	Developers	TestNG, Jenkins
Integration Tests	Developers	TestNG, Jenkins
System Tests	WP37	Sikuli, Jenkins
Acceptance Tests	Task 34.6	Manual Testing

4.1 Unit tests

Unit tests are the lowest level of abstraction and, therefore, need to be created by the developers themselves. They need to make sure that the code they have written works as expected. In order to ensure that the defined roles have the ability to write their unit tests easily, we introduce the tools in chapter 5.2.

4.2 Integration tests

One level above the unit tests follows the integration tests. At this level the testers ensure, that the modules interact accordingly. In order to test an interaction, all involved modules are pulled from the repository and evaluated regarding the interaction with each other. If there is a mistake, or misunderstanding regarding interfaces, errors will appear here.

For the integration tests the developers are responsible again, as they use the interfaces of other modules and have to make sure that they act how they are intended to do.

4.3 System tests

Software often causes problems even if the unit tests and the integration tests are passed. To make sure that the software runs accordingly under real conditions, system tests are essential.

System tests therefore test the software in the actual execution environment the software is planned to run on. The *FutureID*-Client is based on Java™ and must therefore also be tested with different Java™ virtual machines like Open JDK6, OpenJDK7, OracleJDK 6 x64, OracleJDK 7 x64.

4.4 Acceptance tests

Acceptance tests ensure that the software not only is free of bugs, but also is intuitive for real users. This type of tests can't be done automatically, however it is possible to evaluate different metrics, which influence the usability.

In order to make the testbed manageable we won't provide automated acceptance tests, instead, it is the task of WP34 to ensure that the developed user interface is implemented correctly.

5. Test tools

Without the right tools, testing is a very time consuming and resource binding task. Manual testing would lead to a low test-coverage and low rate of testing. In order to be able to test a variety of functions on different levels, it is essential to have the right toolset for test automation and result reporting.

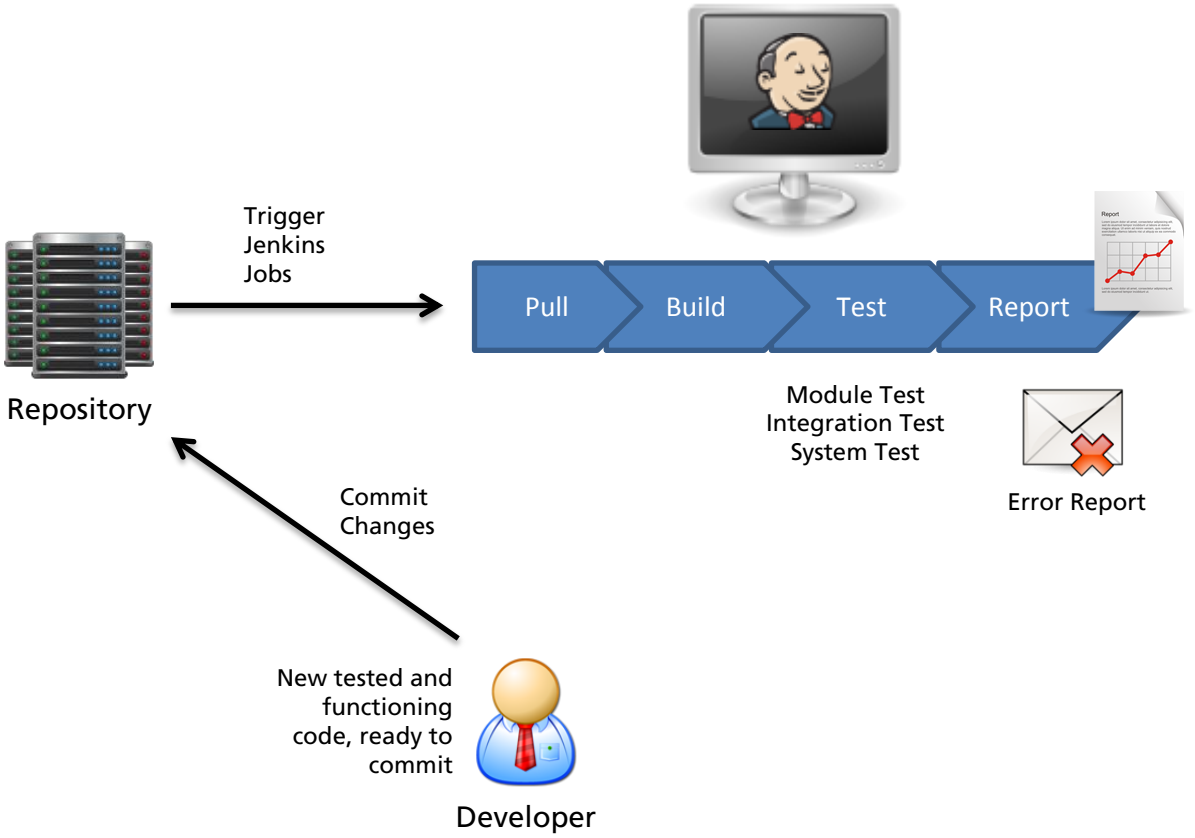


Figure 2 - High level overview of the test tools

As stated in D37.1, the testbed must provide tools to automate tests and to show results on a central platform. In compliance with the requirements for the testbed, the following set of tools has been setup.

5.1 Central continuous integration platform

Jenkins[5] is a continuous integration platform with a modular architecture and the ability to extend the functionality via plugins. It is the main entry point for testing and evaluating test results. Therefore all further tools will be integrated in Jenkins. Furthermore, all test results will be collected and presented on this platform.

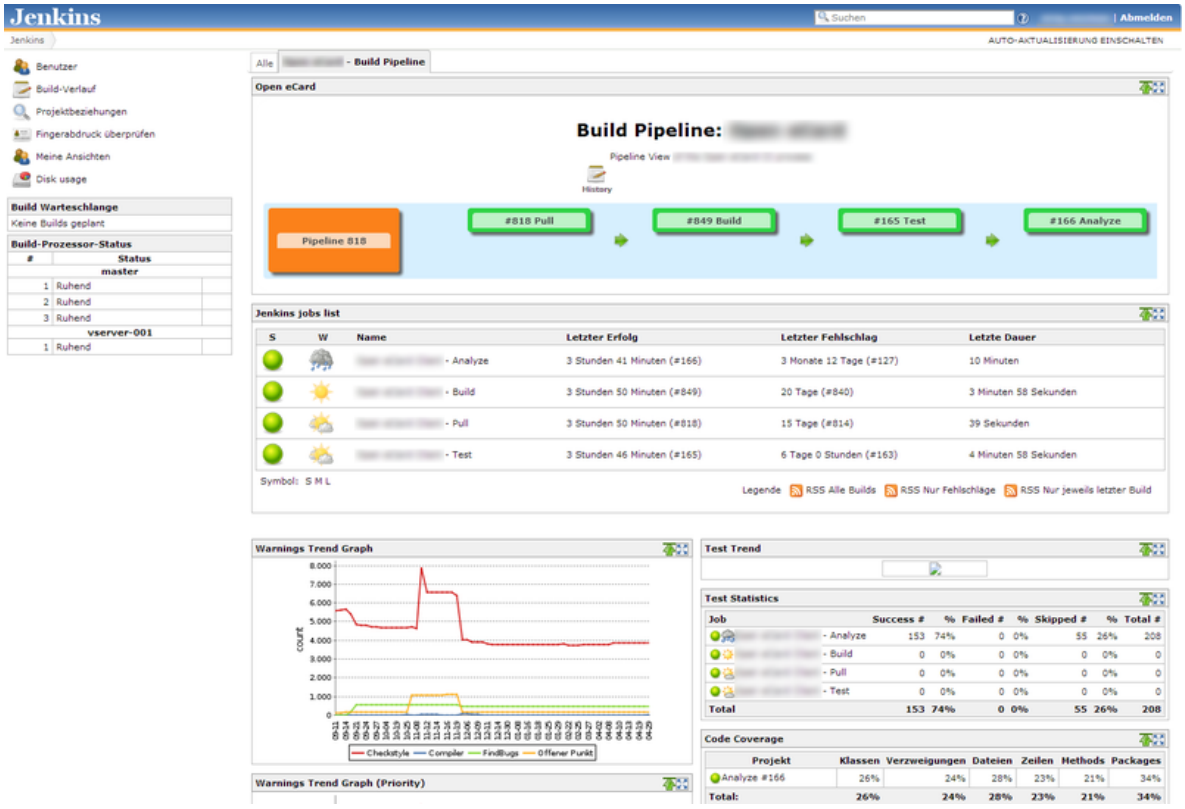


Figure 3 - Jenkins Dashboard

Jenkins provides a summary of all the important information on the main view for all participants. As the build pipeline shows, different stages can be defined as jobs, which can be executed consecutively. This build pipeline reflects the testing process, which was defined in Deliverable D37.1.

[5] http://jenkins-ci.org/

The bottom up approach will be applied using different jobs in Jenkins. First the code will be pulled from the repository, built, and tested and afterwards the code will be analysed.

To test the different levels, different plugins will be integrated into the Jenkins platform, which are then used for the specific tasks.

In the continuous integration platform, different tools will be included to fulfil the different tasks in order to ensure a high quality of the developed software.

5.2 Unit tests

For unit testing TestNG will be used to create unit tests easily. As defined in the requirements, the developers are responsible for creating unit tests. Please make sure, that the code that gets submitted to the repository passes the unit tests that you have created beforehand.

TestNG is a powerful tool that makes the definition of unit tests easy and provides the developers with additional functionalities. One of the most important functionality is that it is easy to integrate TestNG into the continuous integration platform, which executes the tests and presents the reports of the recent tests.

The documentation of TestNG can be found on this website[6]. A detailed tutorial on how to install and use TestNG can be found here[7].

For methods concerning security it can also be useful to check for expected exceptions, like shown here[8].

[6] http://testng.org/doc/documentation-main.html

[7] http://www.asjava.com/testng/testng-eclipse-tutorial-install-and-use-testng-eclipse-plugin/

[8] http://www.asjava.com/testng/testng-tutorials-expectedexception-test/

5.3 Integration tests

Integration Tests will also be developed with the use of TestNG. As you can call methods from other modules and check these methods for exceptions, it is also relatively useful for integration tests.

However, it is planned to divide unit tests and integration tests to have a clear separation between those both levels of testing. The detailed toolset will therefore be published on the wiki later and will be updated throughout the whole project.

5.4 System tests

The next step after integration tests are system tests. The goal is to test the system as a whole, in a near production environment. The view of the tester switches from a developers view to a consumer-oriented view.

The term “system” has no coherent definition. Depending on the view of the test planner this can be either the software alone or software and connected hardware.

In the case of *FutureID*, we should include soft- and hardware into the definition of the system. The reason is mainly because the future software will run on a broad range of hardware. The most common use cases, from the current perspective, seem to be commodity PC hardware, tablet PCs and several kinds of smartphones. The underlying hardware is not necessarily the same. So we propose to include it into system testing.

System testing is one of the most time-consuming test levels, as the software needs to be built, installed and executed in a predefined execution environment. However, there is the possibility to automate many of these tasks.

The building and installation of software can be done automatically by the continuous integration platform. In order to perform certain tasks an additional tool is needed.

As there are different platforms, the *FutureID*-Client needs to be tested on the following chapters will show the tools for the different platforms separately.

5.4.1 Relevant platforms

While currently a majority of users use Microsoft Windows for daily use, it seems, that a shift of operating system use is on the way. Smartphones and tablets are used by a substantial number of users and it is predicted that those numbers will grow further in the future. The Android system was developed by Google and is distributed by a broad range of manufacturers (Samsung, Sony etc.). It is currently the system with most users on a worldwide basis. Due to the open nature of the system it is expected that more hardware manufacturers will use this platform. This leads to an even larger user base. Android is based on GNU/Linux. Hence, it is useful to include this operating system into the system tests.

Furthermore there are several Linux distributions which try to attract desktop users. At least Ubuntu and OpenSUSE have had significant success in this regard. Several German city governments as well as the Spanish region Extremadura made a transition to GNU/Linux. This is a further reason to have system tests for GNU/Linux-based machines.

Microsoft Windows, as well as the several desktop-based GNU/Linux variants, usually run on commodity PC hardware. Currently most new systems use 64-bit architectures. However there is a considerably amount of systems which only support 32-bit. So it is recommended that the system tests support both variants. Furthermore Android devices often use ARM processors. Therefore, this is a third hardware platform to support during system tests.

The Deliverable D37.1 – Testbed Requirements stated the following requirements for system testing as “should”:

- Windows 7 32-bit
- Windows 8 64 bit
- Ubuntu Linux 64 bit
- Macintosh 10.8
- Android 4.1

In the following chapters tools will be presented which will enable automated testing for these platforms.

5.4.2 Windows, Linux and Macintosh

One kind of platform is personal computers, which include operating systems like Windows, Macintosh and Linux. These operating systems are different but there are tools that work with Java™ and therefore can be used on all of these platforms.

We decided to use the tool "Sikuli"[9] for automated system tests, as this tool can run on all mentioned platforms. There are also additional important features that made the decision for the use of Sikuli in this project complete.

Sikuli is an automation tool that works with image recognition (Givens et al. 2013). It is easy to create a script that runs certain tasks and also checks for the right behavior of the tested software. The scripts which are created can then be copied to the other operating systems.

Figure 4 shows the easy to understand scripting language which works entirely with image recognition.

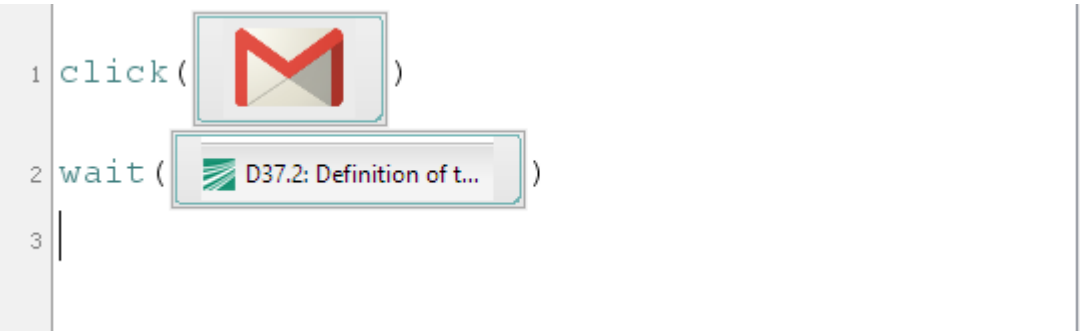


Figure 4 - Sikuli scripting

[9] http://www.sikuli.org/

Also complex user scenarios can be automated with this tool. One exemplary user scenario has been created to test the capabilities of Sikuli during the evaluation of tools. This user scenario includes the following steps:

1. Press “Windows-Key”
2. Type “FireFox”
3. Press “Enter”
4. Click on address bar
5. Type “https://www.ccepa.de/testportal/download.jsp?file=0”
6. Click on “Download mit nPA”
7. Wait for *FutureID* Client
8. Click on “Next”
9. …

This example scenario shows that even cross application automation is no problem with Sikuli image recognition (Chang, Yeh, and Miller 2010). The scenario starts with native Windows 8 actions, opens the browser, navigates to a service provider, and navigates through the *FutureID*-Client.

Furthermore it is possible to integrate Sikuli scripts into the continuous integration platform Jenkins and run these batch files, to test on a frequent basis.

It is planned to identify different service providers and to test the *FutureID*-Client under real conditions with already existing services. This will lead to a high rate of testing of the defined user workflows and will ensure that the *FutureID*-Client always acts as intended.

A detailed tutorial on how to use Sikuli and how to integrate the Sikuli scripts into Jenkins will be provided on the wiki. It is highly recommended for all participants, which are concerned with system testing, to have a look at this tutorial and run as many tests as possible with the use of Sikuli.

Manual system tests should be avoided for the most part as it is time consuming and not as reliable and reproducible as automated tests.

5.4.3 Android

As Android is also one of the designated platforms for *FutureID*, system tests for this platform will be performed as well. There are several challenges for identifying the right toolset to automate android tests. It is not as easy as for desktop computers to run the Jenkins platform on it and execute the tests.

In order to make sure android devices can be tested automatically as well, the devices need to be plugged in to a personal computer, which is running Jenkins. This device needs to be installed and configured on the personal computer, to make sure that it will be identified by the operating system. With the use of Jenkins and distributed slave machines it is possible to achieve this task.

There are different tools to define test steps and run black-box tests on different installed applications. The most promising and sophisticated tool is Robotium[10], which is licensed under the Apache License 2.0.

Robotium allows the execution of system tests, which can be used to also test cross-application scenarios, which are needed in the *FutureID* context.

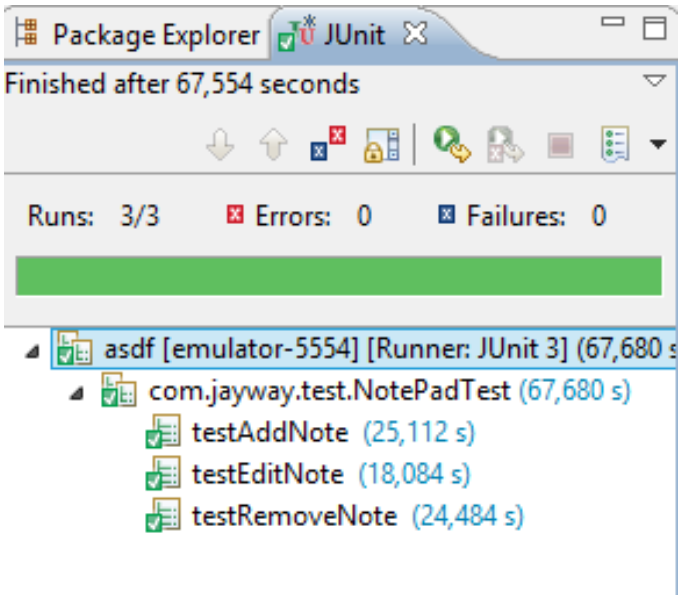


Figure 5 - Test results

[10] Further information on: https://code.google.com/p/robotium/

An example for such a test case is shown in Figure 6. It is possible to create assertions, run applications, enter text and do other kinds of native android actions like button presses and the configuration of timings. Furthermore it is possible to take screenshots at certain steps within a test case, to make sure that, for example, a specific application screen looks the same on different devices[11].

NotePadTest.java | NotePadTest.java

```
}

public void testAddNote() throws Exception {
    solo.clickOnMenuItem("Add note");
    //Assert that NoteEditor activity is opened
    solo.assertCurrentActivity("Expected NoteEditor activity", "NoteEditor");
    //In text field 0, enter Note 1
    solo.enterText(0, "Note 1");
    solo.goBack();
    //Clicks on menu item
    solo.clickOnMenuItem("Add note");
    //In text field 0, type Note 2
    solo.typeText(0, "Note 2");
    //Go back to first activity
    solo.goBack();
    //Takes a screenshot and saves it in "/sdcard/Robotium-Screenshots/".
    solo.takeScreenshot();
    boolean expected = true;
    boolean actual = solo.searchText("Note 1") && solo.searchText("Note 2");
    //Assert that Note 1 & Note 2 are found
    assertEquals("Note 1 and/or Note 2 are not found", expected, actual);

}
```

Figure 6 - Robotium Testcase

These types of tests will also run within a Jenkins job on a certain basis. It is too early to specify a timeframe in which the tests will be run periodically. However a suggestion is to run system tests on a daily basis, as they are time consuming.

[11] The user interface can then be evaluated as well to ensure user acceptance.

5.5 Acceptance tests

Acceptance tests are tests that evaluate if the software is intuitive for the designated user groups. As the results of the acceptance tests are bound on the feeling of the users, an automation of this type of tests is not possible at the moment. However the most recent version of the applications will be provided on the Jenkins platform to ensure, that always the most recent version will be tested by the users.

Additional tools will be evaluated during the lifetime of the project.

6. Test documentation

Deliverable D23.2 was concerned with documentation guidelines for the *FutureID* project and also specified test documentation guidelines to create recommendations for the testbed.

6.1 Test cases

The general character of test cases will be explained in this chapter. In order to have a common understanding of the test case documentation a template will be provided within this chapter.

The requirements for the *FutureID*-Client have been developed within the tasks T31.1, T32.2, T33.1 and T34.1. These requirements should later on be used to derive test cases for the *FutureID*-Client in order to test whether the client fulfils the specified requirements.

The following information is needed:

- Test case id
- Test case description
- Related requirements
- Depth
- Test category
- Author
- Automated?
- Pass/fail
- Remarks

For the documentation of the developed test cases a wiki page will be set up to have a central and always up to date summary of all test cases.

6.2 Test results

The use of the continuous integration platform enables the possibility to create and present reports on the driven tests. This will be used to provide a transparent way to show the state of the software with different metrics.

6.2.1 Build success

Some errors can cause that the developed software can't be compiled, which will lead to a build error. These errors need to be fixed immediately, as they block the whole functionality of the developed software.

In order to have an overview of the successes and fails of the latest builds, a graph will be provided within Jenkins. In Figure 7, an example of builds and their success is shown.

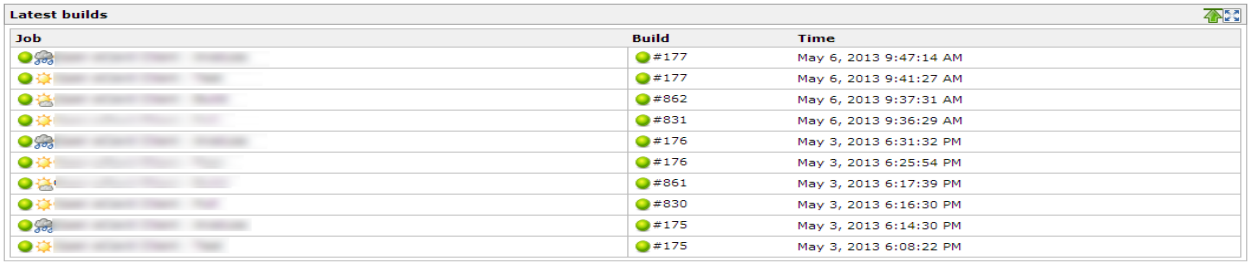
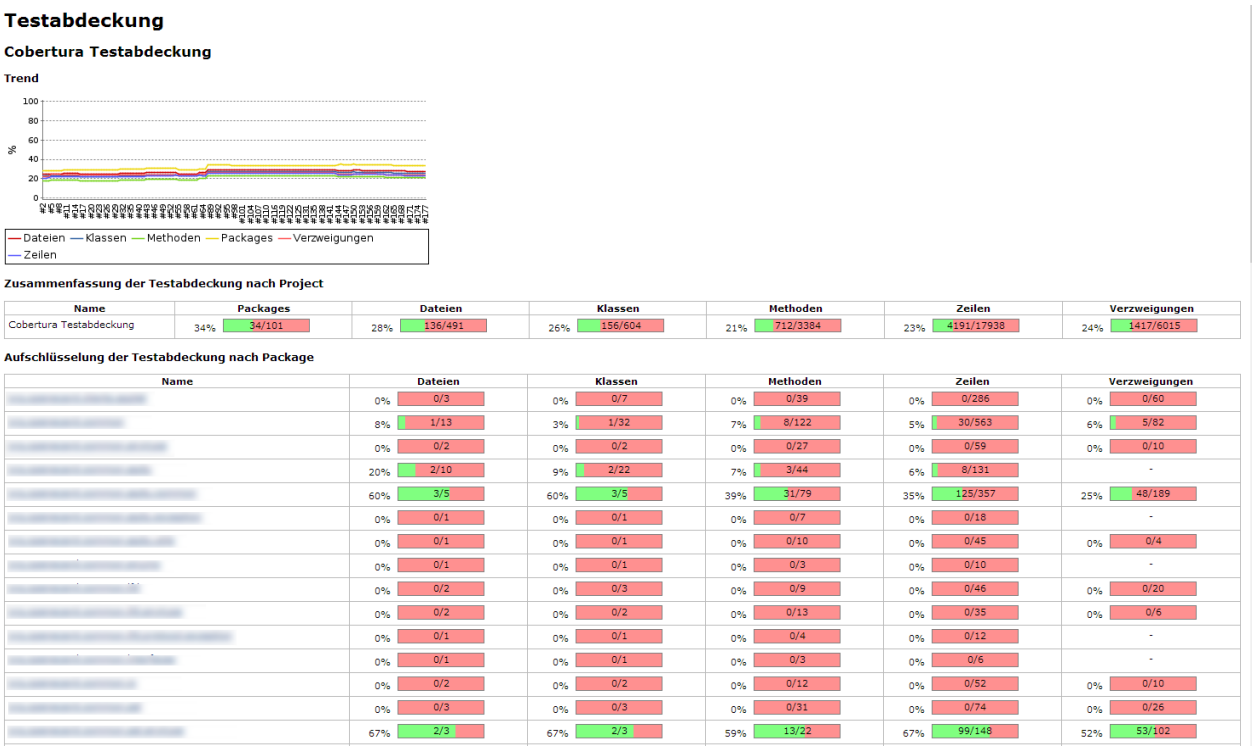


Figure 7 - Build report

6.2.2 Code coverage

For the evaluation of test coverage the tool "corbertura"[12] will be used. Figure 8 shows an example of the test coverage report with corbertura.



Testabdeckung

Cobertura Testabdeckung

Trend

Zusammenfassung der Testabdeckung nach Project

Name	Packages	Dateien	Klassen	Methoden	Zeilen	Verzweigungen
Cobertura Testabdeckung	34% 34/101	28% 136/491	26% 156/604	21% 712/3384	23% 4191/17938	24% 1417/6015

Aufschlüsselung der Testabdeckung nach Package

Name	Dateien	Klassen	Methoden	Zeilen	Verzweigungen
[redacted]	0% 0/3	0% 0/7	0% 0/39	0% 0/286	0% 0/60
[redacted]	8% 1/13	3% 1/32	7% 8/122	5% 30/563	6% 5/82
[redacted]	0% 0/2	0% 0/2	0% 0/27	0% 0/59	0% 0/10
[redacted]	20% 2/10	9% 2/22	7% 3/44	6% 8/131	-
[redacted]	60% 3/5	60% 3/5	39% 31/79	35% 125/357	25% 48/189
[redacted]	0% 0/1	0% 0/1	0% 0/7	0% 0/18	-
[redacted]	0% 0/1	0% 0/1	0% 0/10	0% 0/45	0% 0/4
[redacted]	0% 0/1	0% 0/1	0% 0/3	0% 0/10	-
[redacted]	0% 0/2	0% 0/3	0% 0/9	0% 0/46	0% 0/20
[redacted]	0% 0/2	0% 0/2	0% 0/13	0% 0/35	0% 0/6
[redacted]	0% 0/1	0% 0/1	0% 0/4	0% 0/12	-
[redacted]	0% 0/1	0% 0/1	0% 0/3	0% 0/6	-
[redacted]	0% 0/2	0% 0/2	0% 0/12	0% 0/52	0% 0/10
[redacted]	0% 0/3	0% 0/3	0% 0/31	0% 0/74	0% 0/26
[redacted]	67% 2/3	67% 2/3	59% 13/22	67% 99/148	52% 53/102

Figure 8 - Corbertura

This tool shows a report on the test coverage. An overall test coverage is shown as well as the coverage of different classes.

[12] http://cobertura.sourceforge.net/

7. Success factors

There are some general factors and criteria which need to be considered in order to arrive at a successful testing process and subsequently at a high quality software product.

Coverage: The amount of code that is covered by tests should be as high as possible.

Automation: The amount of tests that can be automated should be as high as possible.

Accessibility: The testbed and its tools should be accessible and easy to use, in order to facilitate testing activities for all responsible parties.

Entry criteria: Before testing can begin there may be certain requirements that first have to be met in order to proceed. These criteria should be clearly defined beforehand. These could, for example, be requirements like:

- Other testing activities have to be completed or signed off first.
- Systems and resources involved in the test have to be available and/or have to have a certain status

Exit criteria: There should be a clear definition of requirements to be met in order to consider a testing activity as completed. This could for example be:

- A certain percentage of code has to be tested successfully.
- High priority defects have been fixed.

8. Conclusion

The *FutureID*-Client is a complex application with cross-application workflows, which makes a well thought-out testbed necessary. Especially system tests, which should run periodically to ensure that all functions are still working properly, are very time consuming.

For this reason this deliverable concentrated on setting up a suitable testbed with different kinds of test tools, which are all integrated into the central continuous integration tool Jenkins. Each of these tools fulfils a certain task within the big objective of thorough testing of the *FutureID*-Client.

9. Bibliography

Chang, Tsung-Hsiang, Tom Yeh, and Robert C. Miller. 2010. “GUI Testing Using Computer Vision.” In *Proceedings of the 28th International Conference on Human Factors in Computing Systems*, 1535–1544. http://dl.acm.org/citation.cfm?id=1753555.

Givens, Paul, Aleksandar Chakarov, Sriram Sankaranarayanan, and Tom Yeh. 2013. “Exploring the Internal State of User Interfaces by Combining Computer Vision Techniques with Grammatical Inference.” Accessed May 7. http://www.cs.colorado.edu/~srirams/papers/icse2013-nier.pdf.