Library Management System

Test Management Concept

CANDIDATES NAME: MATTA KIMANI

REG. NO.: 20/03569

SUBMISSION DATE: 21/10/2022

SUPERVISSOR: COLLINS ONDIEK

COURSE: BACHELOR OF SOFTWARE DEVELOPMENT

Contents

0. Introduction

Based on a fictitious software development project [1] the task for this term paper is to provide a detailed proposal for the test management part. The primary goal is to evaluate and to propose a set of tools which can support most of the test activities of the project.

To make the exercise more realistic and to make the competitive situation more concrete the paper starts with a chapter describing assumptions which complement the description [1] given.

Starting point for further considerations will be the analysis of the fundamental test process which was presented during the lecture [2] followed by the description of all test activities within the test process.

The remaining chapters focus on the evaluation of possible tools including discussion of implemented prototypes, advantages and limitations, interfacing and reporting capabilities as well as decision making support.

The paper ends with cost-benefit considerations and possibilities for integration.

The system under test (SUT) is a library management system (LMS) which provides aweb-interface to access the library. LMS consists of two databases:
- database for library users (names, addresses, ID numbers, borrowed books)
- database for librarians (book titles, authors, number of copies, availability status)
The functionality of the software includes the usual services like inserting, deleting, updating and searching of information. [1]

1. Assumptions

The project description received [1] provides a good overview about the project. However in a real situation we would query additional details to ensure that the approach is properly tailored. This is why we work out various assumptions which are described in this section.

1.1 The SUT technology stack
We assume that the SUT is traditional web development using following 3-tier architecture:
- web server: Apache Tomcat
- application server: Java EE
- DBMS: mySQL

1.2 The SUT development environment
Is based on JEE 7 with Spring Framework 4.2.4. IDE = IntelliJ IDEA

1.3 System architecture
All servers are virtualized and the preferred operating system is Redhat Enterprise Linux (RHEL). The customer currently runs four different environments:
A. development environment
B. test environment
C. User Acceptance Test (UAT) environment ("near production" environment)
D. production environment

A and B are only used by the technical teams.
C is the designated environment for end user testing.
D is basically not available for testing.

1.4. Customer

The customer belongs to a governmental organization. The budgets are limited. This is why there are for the moment no plans to acquire commercial software licenses for software testing purposes.

1.5 Software testing experience

We assume there is no specific experience available on customer side concerning software testing. However, the customer plans to further develop the product and software testing is regarded as key success factor. Therefore standards as well as best practice recommendations are highly welcome.

1.6 Testing tools

As mentioned before the customer has no budget für software licensing. But we assume the customer is willing to provide budget for implementation and configuration of proper tool support. This is why we will consider only open source tools which are available for free or can be licensed with minor costs. Furthermore we suppose that a test framework which allows a high integration of various tools is preferred. Furthermore we assume that the integration of testing tools into IntelliJ IDEA, the preferred development environment, is welcome.

2. Analysis of fundamental test process

2.1 Testing process versus software development process

Our test approach for this project shall follow the fundamental test process of the International Software Testing Qualifications Board (ISTQB). The process is a generic approach and will be discussed in the following sections in more detail.
Important to note is the fact that the testing process is heavily dependent on the software development process. In our case we have to cope with a combination of linear sequential model ("waterfall") and prototyping model. [1]

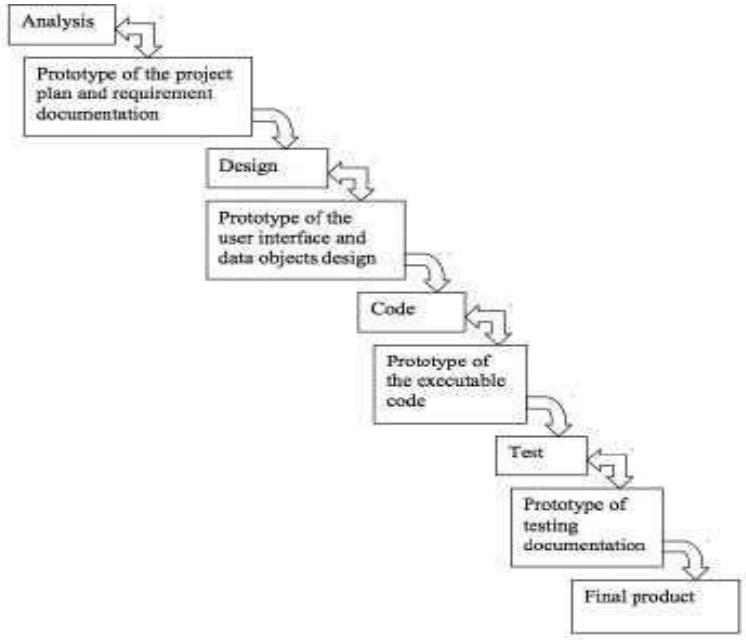

Figure 1: Software Process Model and Quality Assurance involvement [1]

This is also why the SUT is planned to be delivered in 4 cycles:
- Release v0.1beta
- Release v0.9beta
- Release v1.0
- Release v1.0alpha

Although we discuss in the fundamental test process in this chapter step-by-step, the following chapters will adapt the process to the individual requirements of the project. Meaning, we do not follow strictly the generic sequence but allow steps to take place simultaneously or overlapping.

2.2 Standards

Before we elaborate on the generic test process itself, it is worth to discuss with the customer the use of standards. Basically would prefer to apply standards throughout the test process as much as possible. This can be achieved on multiple levels. [4]

a. Company standards

Although there is limited time to establish and also align internal company standards, we would urge the customer to establish a top level document for software testing in the organization aka "Software Testing Policy". This would serve as guideline for all testing activities in the organization. The initial content could include [2] principles of testing, organizational specifics, generic test process definition and description of an approach to improve the test processes continuously. For example obligatory meetings at the end of each project or testing activity to reflect on lessons learned. Furthermore, the preferred set of testing tools can be specified to avoid uncontrolled growth in this respect.

b. Best practices

c. Standards for particular industrial sectors

b. and c. both strongly depend on the experience of the testing team as well as the requirements of the customer and should be aligned among the stakeholders up-front.

d. Software testing standards
Although compliance to such standards may not be required by the customer, to obey standards may have positive impact, e.g. by standardization of terms and definitions. Furthermore the test team can score points not only with technical know-how or innovative approaches but also by compliance to market standards.
Exemplary list:
- IEEE 1028 generic process for formal reviews
- ISO/IEC/IEEE 29119-1: Concepts & Definitions
- ISO/IEC/IEEE 29119-2: Test Processes
- ISO/IEC/IEEE 29119-3: Test Documentation
- ISO/IEC/IEEE 29119-4: Test Techniques
- ISO/IEC/IEEE 29119-5: Keyword Driven Testing

In this respect we would definitely consider *IEEE 829-2008, also known as the 829 Standard for Software and System Test Documentation.*

Throughout the project various testing related documentation will be needed. Without a structured approach with standardized documents and formats, effectiveness will suffer.

With reference to the IEEE standard 829 [2] the concerned documents can be categorized as follows:

Figure 2: The software test documentation pyramid

As mentioned before all testing related documentation shall be lead by an company-wide policy. Underneath you define one test strategy which describes the approach on project level, followed by several test plans, one plan for each test level. "Test reporting" refers to documents which are created as needed in the course of the project like anomaly or error reports, level test reports summarizing test results or test design documents describing test cases, expected results, entry or exit criteria and so on.

2.3 The fundamental test process
The figure 3 shows the fundamental test process which is triggered by the test strategy as well as the test plan. It is an iterative process which is repeated for each test level respectively in our example also for each release cycle.
The testing effort ideally decreases over time due to a steep learning curve. The preparation and initiation for the initial release is much higher compared to the subsequent releases.

Furthermore the goal of an efficient test process is to detect defects as early as possible (e.g. during beta testing). Costs for fixing a defect at a later stage is much more expensive compared to an early stage. This will finally also allow limited user testing (LUT) at later stages, focusing only on a relative small number of outstanding errors (also called regression testing).

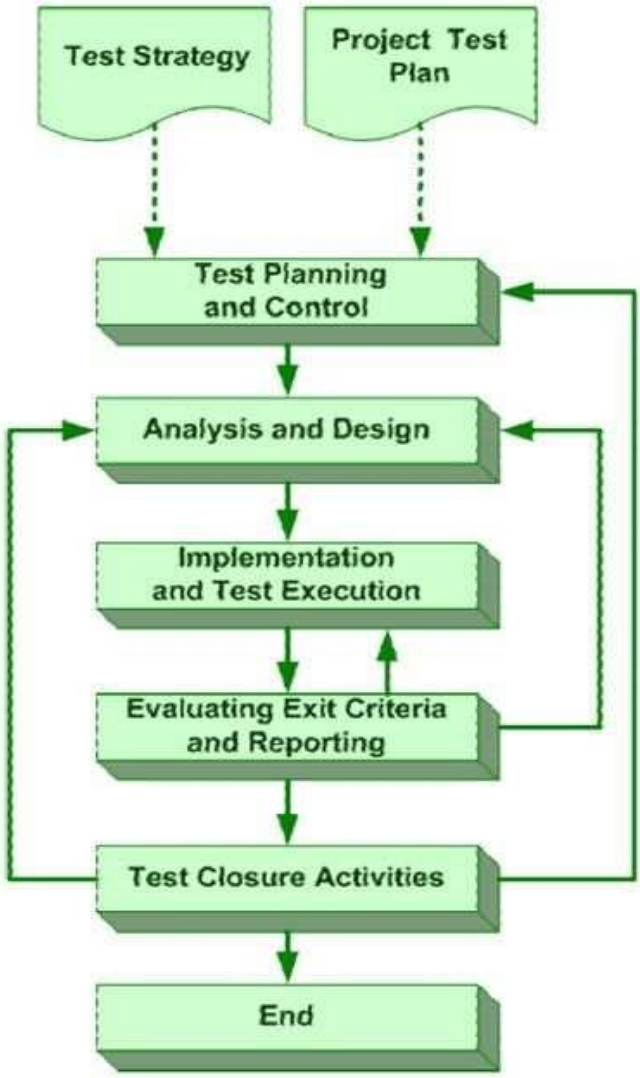

Figure 3: The fundamental test process [2] [4]

2.4 Test Planning and Control

At the beginning of the repetitive test process there is a plan which influences respectively steers the following stages of the process. At the same time a control process escorts all sequences of the plan to allow target-performance comparison. This will lead to ongoing adaptations and improvements.

The planning happens as mentioned on different levels and can serves at the end of each test activity as input for the next planning effort (feedback loop).

Major tasks of test planning. [2]

Determine the scope and risks and identify the objectives of testing

The objective is to proof that the delivered functionality is matching the specified requirements (see requirement documentation)

The system functions [1] are described as follows. We differentiate between the main users, the librarians and the library users.
Librarian operations [1]
--login to the system
--add, modify or remove information about the book to/from the book database
--check availability of the book using books call number
--add, modify or remove information about the library users to/from the users database
--view the list of library users
--generate report, containing the information about users of library, and books they overdue
--search the book by author or book title
Library users [1]
--login to the system
--search the book by author or book title
--view the information about his/her status
--renew taken book through the Internet using web-based interface
Furthermore we have to consider non-functional requirements like usability (user interface), performance and reliability.
The main testing scope is outlined [1] as follows
- functional testing including validation testing and specific functional testing
- Integration testing
- User Acceptance Test
- Performance testing
- Bash Multi-User testing
- Technical testing

Finally, the identification of risks and definition of countermeasures need to be evaluated.
In this specific project this could be
- the risk of insufficient integration of the external testing team into the overall project
- the risk of availability of skilled team members or
- the risk of unstable requirements resulting in many change requests
The latter eg could be covered with a consistent change management procedure.

Determine the test approach
The goal is to choose a test approach that optimizes the relation between cost of testing and cost of possible defects as well as minimizing the risks. [4]
We would suggest a preventive approach where the test team is involved in the development process as early as possible and escorts development process in all phases. Furthermore we would suggest a risk based testing prioritization meaning we focus on test objects which are more important and complex (more risky) instead of unimportant aspects with low risk.

Implement the test policy and/or the test strategy
As mentioned in 2.2 a test policy would be helpful. The test policy as well as the principal approach from before will be the base for the test strategy.
Following test level are defined in the project:
--Unit module testing
--Integration testing
--System testing
--User acceptance testing (UAT)

Determine the required test resources like people, test environments, PCs, etc.
A crucial part of test planning are the required resource:

- Human test ressources
It is assumed that 4 persons (4 FTEs, full time equivalents) are needed.
We would create a job description for each position and allocation tasks and responsibilities.
This will help in the selection process to identify the right staffing and to reveal training needs.

- Organisational structure with test management
Adequate test organization needs to be foreseen. In our case we assume a dedicated testing team lead by a test manager who is reporting to the overall project manager.

- Test infrastructure
This is comprising the workstations of the test team including all nodes required to perform the tests. In our case the development, test and UAT servers.

- Testware
Primarily we focus here on the selected testing tools. But generally speaking this includes scripts, setup or cleansing procedures, config files, databases and so on, simply any software or artifact needed for the test process.

Schedule test analysis and design tasks, test implementation, execution and evaluation
To allow measurement of progress all test activities shall follow a predefined schedule.
Assignment [1] defines release schedule and test process schedule (see GANTT chart).
Still missing is the allocation of resources to show exact usage of resources.

Determine the exit criteria, therefore we need to set criteria such as coverage criteria
Due to limited resources exit criterias need to be defined e.g. test coverage which is a preferred criteria. This helps to measure if a test activity was successful. This essential due to the phased approach where exit criteria represent often entry criteria for the following phase.
The assignment demands. [1]
- all code must be unit tested
- unit and link testing sign-off by development team
- test infrastructure must be ready and available for system testing
- acceptance tests must be completed with pass rate > 90%
- all High Priority errors from System Test must be fixed and tested

Major tasks of test control. [2]

Plan-do-check-act (PDCA) is the underlying step-by-step method to continuously improve the testing process. Therefore "check" or "control" are imperative.

Measure and analyze the results of reviews and testing
All test results need to be benchmarked to allow judgment of testing efficiency.

Monitor and document progress, test coverage and exit criteria
Provide information on testing
All test activities including results need to be monitored and the progress needs to be documented and communicated to all stakeholders. This includes assessment of the test coverage achieved and comparison with the corresponding testing exit criteria.

initiate corrective actions
This concerns the defects or anomalies identified during testing which results in creation of trouble tickets in the incident- or configuration management tools. And also inefficiencies of the process itself need to be reported to test or project management to initiate amendments.

Make decisions
Last but not least, test control demands various decision making. Based on the test progress and its results eg the test manager has to "release" a software version or testing phase. Or testing needs to be suspended to allow the development to provide major improvements. Or the the testing need to be suspended due to insufficient testing progress which can result in a revision of the test plan.

2.5 Test Analysis and Design
Based on the initial planning or based on concrete results of already executed or closed testing activities, test analysis and and test design [2] takes place.

Review the test basis
What should be tested needs to be understood first. Therefore we need to review relevant technical specifications or architecture documents [4], risk assessment papers or any other documentation available.

Identify test conditions
Based on the requirements, the conditions need to be determined to allow proper setup of the tests. What kind of test data will be required? What is the initial state (before testing)? What is the expected or desired outcome?
This will finally support traceability between input and output and allow reproducibility.

Design the tests
Here comes the creative part of choosing the right technique for testing software.
Basically it is distinguished between static techniques like structured evaluations or reviews or static analysis without execution of the program.
And there are dynamic techniques where the programs are executed. Following the categorization in [4] we have black box testing, white box testing or experience based testing.
In this respect we have to start with designing logical test cases which can be translated in the implementation phase into concrete physical test cases.

Evaluate testability of the requirements and system
It goes without saying that it the SUT has to support testability. For example interfaces need to be open and well described. If not, maybe techniques like white box testing will not be feasible. Considering static code reviews demand availability of the source code, and so on.

Design the test environment setup and identify and required infrastructure and tools
Here we define the specific test environment including configuration, hardware (servers, clients), software with proper version, testing tools, etc.

2.6 Test Implementation and Execution
Based on the test design, finally we have to implement and verify the test environment and we have to execute the tests as soon the SUT will be available. In our case four test cycles are planned (v0.1beta, v0.9beta, v1.0 and v1.0alpha)
The beta releases will deliver all functionality and will be the main target for testing.
v1.0 will bring no new functionality and therefore only limited user testing will be performed. v1.0alpha will be only available for bug fix tests only.
Due to the fact that the time frame between the cycles is relative short (just a few weeks) the test preparation and setup needs to be ready before delivery of initial release. And test automation will necessary to meet the tight schedule.

Major tasks of test implementation. [2]

Develop and prioritize test cases by using techniques and create test data for those tests
Concrete test cases need to be derived from design, ideally from logical test cases.
In practice it is often an intuitive process to develop physical test cases. The success highly depends on skills of tester, therefore a systematic approach should be preferred.
Prioritization of test cases is essential. Important test cases should be executed first. This allows tackling of important defects in an early stage and it allows "risk-based testing" [4]
Moreover, proper test data need to be created.

Write some instructions for carrying out the tests which is known as test procedures
Test procedures allow efficient and proper execution. Furthermore it facilitates reproducibility.

Automate some tests using test test harness and automated test scripts
A high test automation brings many advantages. It has a positive impact on test coverage, simply because number of tests can be maximized and is very efficient in case of regression testing. This is needed in two release cycles within the project, but it can be reused also during maintenance phase of the LMS. One drawback is the fact that automated test scripts need to be prepared upfront which assumes these tests are reused. Furthermore test harness is needed. In the early stages of testing not all components will be available, therefore simulators, drivers, dummies or mock-ups (offer more functionality than dummies) are needed.

Create test suites from the test cases for efficient test execution
Test cases are sets of conditions, whereas test suites represent a collection of test cases.

Implement and verify the environment
The test environment itself needs to be verified to minimize the risk that new errors occur based on the fact that the SUT runs in a (artificial) test environment with contains the SUT plus testware.

Major tasks of test execution. [2]

Execute test suites and individual test cases following the test procedures
Prioritization is essential as mentioned before.

Re-execute the tests that previously failed in order to confirm a fix (confirmation testing or re-testing) This is important because a fix can cause new problems.

Log the outcome of the test execution and record the identities and versions of the SUT
"Tests without a log are of no value". [4]
For the subsequent processing of the results a test log is imperative. Comparison of results, reporting of anomalies and reproducibility require proper logging.

Compare actual results with expected results and report discrepancies as incidents
This concludes the process stage of test execution

2.7 Evaluation exit criteria and Reporting

Major tasks evaluation exit criteria and reporting. [2]

Check the test logs against the exit criteria specified in test planning
Continuous evaluation of the exit criteria
Assess if more test are needed or if the exit criteria specified should be changed
Write a test summary report for stakeholders

2.8 Test Closure Activities

Major closure tasks. [2]

Check which planned deliverables are actually delivered and ensure that all incident reports have been resolved
Finalize and archive testware such as scripts, test environments, etc. for later reuse
Handover the testware to the maintenance organization
Evaluate how the testing went and learn lessons for future releases and projects

3. Test activities and tooling

This chapter will elaborate on following aspects
- Description of all test activities within test process
- Evaluation and reasoning of possible tool support including
- implementation of prototypes for proposed tools

Introduction
The main purpose of the project is to test the front end in the first two releases (v0.1beta and v0.9beta). During this test process approximately 90% of all errors would be covered and fixed. In the Release v0.1beta test plan include Acceptance, Functional, Performance and User Acceptance testing. In the Release v0.9beta, as shown in Fig. 4, test plan will consist of Integration, Acceptance, Bash & Multi-User testing, Technical, User Acceptance and Regression testing. Regression testing of outstanding errors will be performed on an ongoing basis. When all errors would be fixed and tested agaIn that means that the system works in integrated manner and the Release v0.9beta is the final, proving of the system as a single application that implement and fulfill the policy and strategy for testing. [1]

The planned release versions which drive the system testing cycles:

Testing by Phase	
Release v0.1beta	Acceptance 1 Functional 1 Performance 1 User Acceptance 1
Release v0.9beta	Integration 1 Acceptance 2 Bash & Multi-User Testing 1 Technical 1 User Acceptance 2 Regression 1
Release v1.0	Installation Test LUT 1 Technology Compliance Testing LUT 1 Bash & Multi-User Testing LUT 1 Regression LUT 1 Regression LUT 2
Release v1.0alpha	Per Bug Fix Test Only

Figure 4: System testing cycles

Tooling

Before we focus on the tools proposed in the specific test levels
- component testing
- integration testing
- system testing
- acceptance testing

we have a look on general tools which we need to support the testing process and consider the testing framework we want to use. The framework consists of a test management tool, a test automation component which acts as integration point for the testing toolset and an integrated defect tracking tool.

General tools

We assume **MS Project** is used for overall resource planning and work breakdown structure (WBS). This means we derive the primary planning information from a MS project plan which is maintained by the overall project manager. This is also the primary tool for maintaining planned and actual effort.

Open Office shall be the tool to create the documentation which cannot be derived automatically from any testing tool.

This includes test policy, test strategy documents, test design, ...

GitHub

We assume the development team make use of a collaborative versioning tool like www.github.com. This could be shared with the testing team. This would facilitate not only access to the software components of the SUT but could be also used as document repository with version control and wikis.

Test Management Tool and Defect Tracker

The central tool for **planning**, **control** and also **reporting** shall be the test management tool. The tool is ideally also linked to the tools for analysis and design and also implementation and test execution. This is why we created a shortlist with possible open source solutions.

One option was www.testlink.org

Unfortunately we lost quickly the confidence into this product. We could not find valuable documentation and also the demo was not working.

Therefore it was easy to decide for www.squasthtest.org

A product which offered diverse documentation and we had only minor technical problems to install the components (see also appendix).

*Squash [5] is an **open source project** aiming to structure and industrialize functional testing activities. These deliverables cover all functional qualification processes:*

- *manual testing (test repository management)*
- *functional test automation*
- *management of test datasets*

The Squash TM (Test Management) features are:
- Multi-project repositories with milestones
- Requirement management
- Test case management
- Run management
- Bug Tracking
- Monitoring and reporting
- Collaborative work

The following screenshots describe some aspects more in detail:

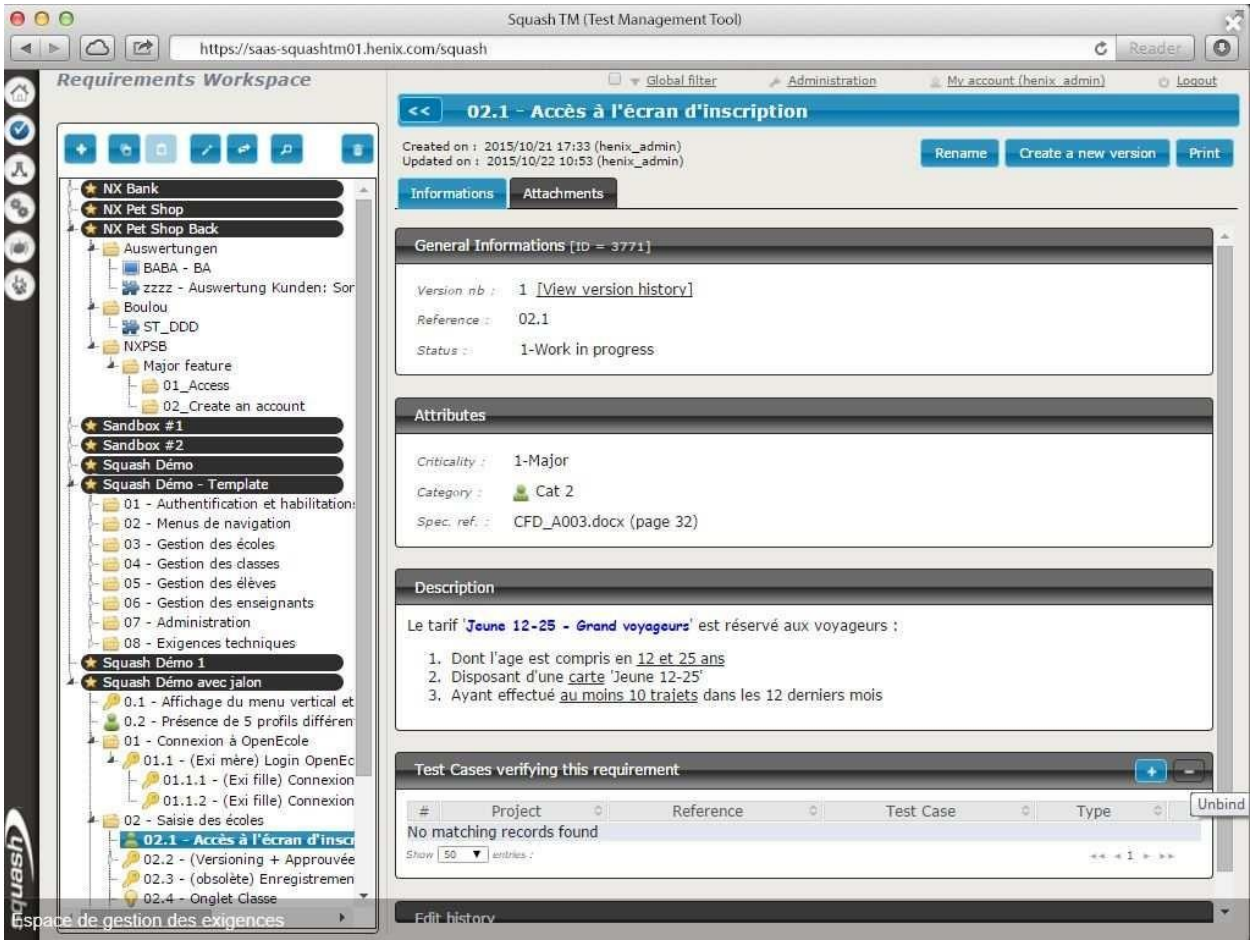

Figure 5

Requirements workspace: the module which supports [5]
- documentation of determined scope and risks including objectives of testing
- requirements can be linked to test cases and test steps can be linked to requirements
- requirements workflow, versioning, logging, printing
- requirements tree (bulk editing, export, ...)

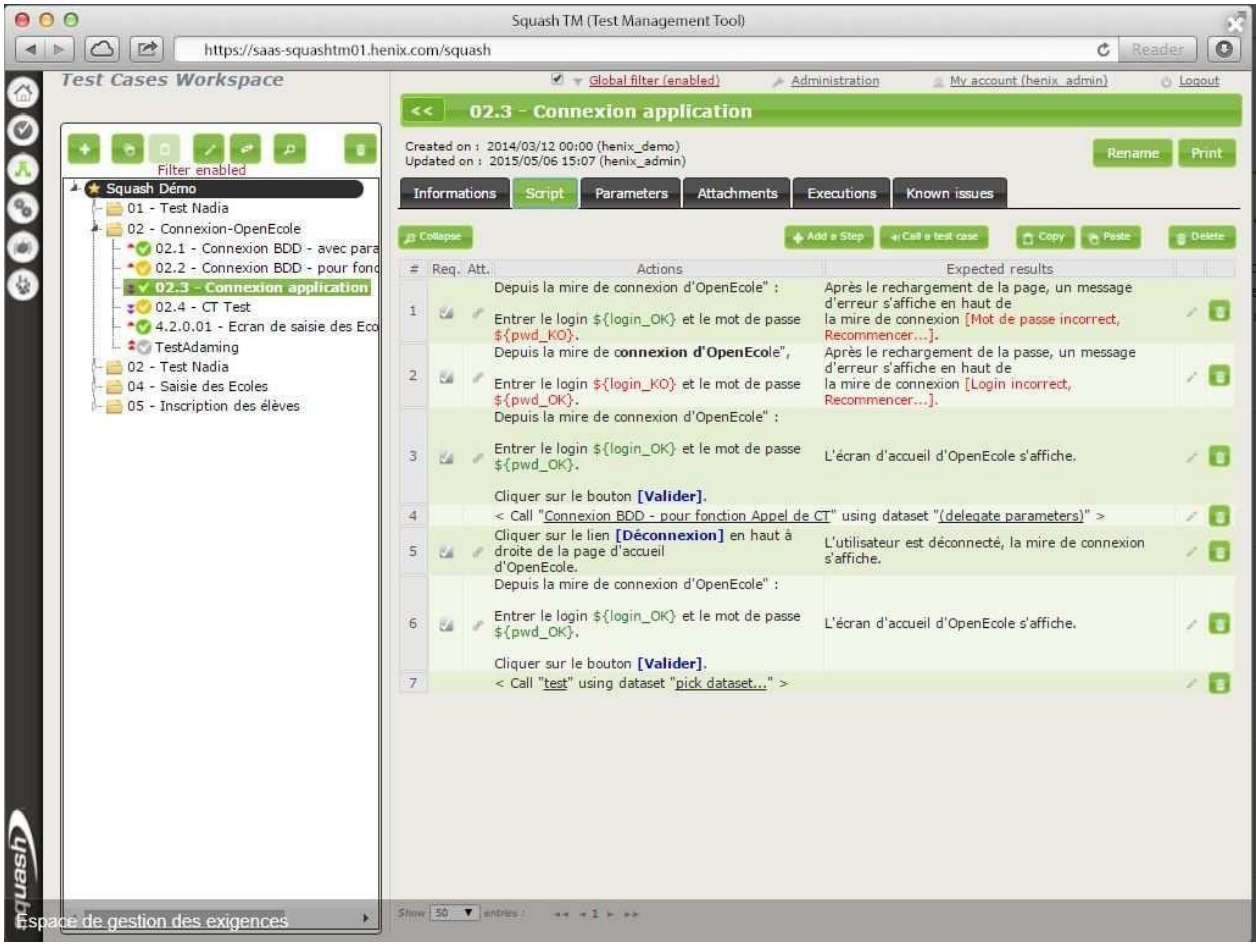

Figure 6

Test cases workspace: the module which supports [5]
- development and prioritization of test cases
- writing of instructions for carrying out the tests (test procedures)
- creation of test suites from the test cases for efficient test execution (test scenarios)
- test case library

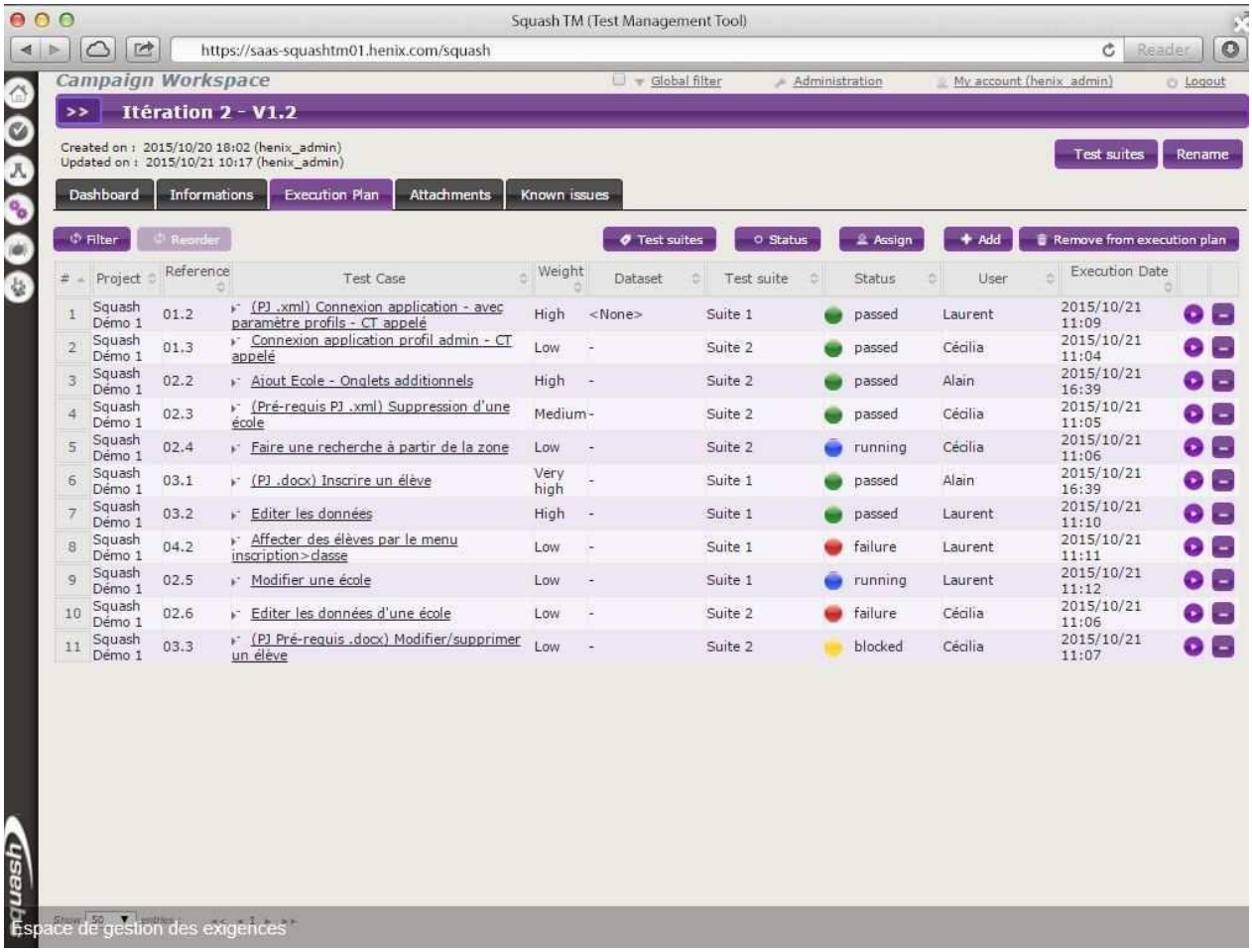

Figure 7

Campaign workspace: the module which supports[5]
- monitoring and documentation of progress, test coverage and exit criteria
- Execution of test suites and individual test cases following the test procedures
- Re-execute the tests that prefiously failed in order to confirm a fix
- log the outcome of the test execution and record the identities and versions of the SUT
- compare actual results with expected results and report discrepancies as incidents

Figure 8

Management workspace: the module which supports [5]
- measurement and analysis of the results of reviews and testing
- reporting libraries
- custom graphs, dashboards, with wizards
- standard dashboards

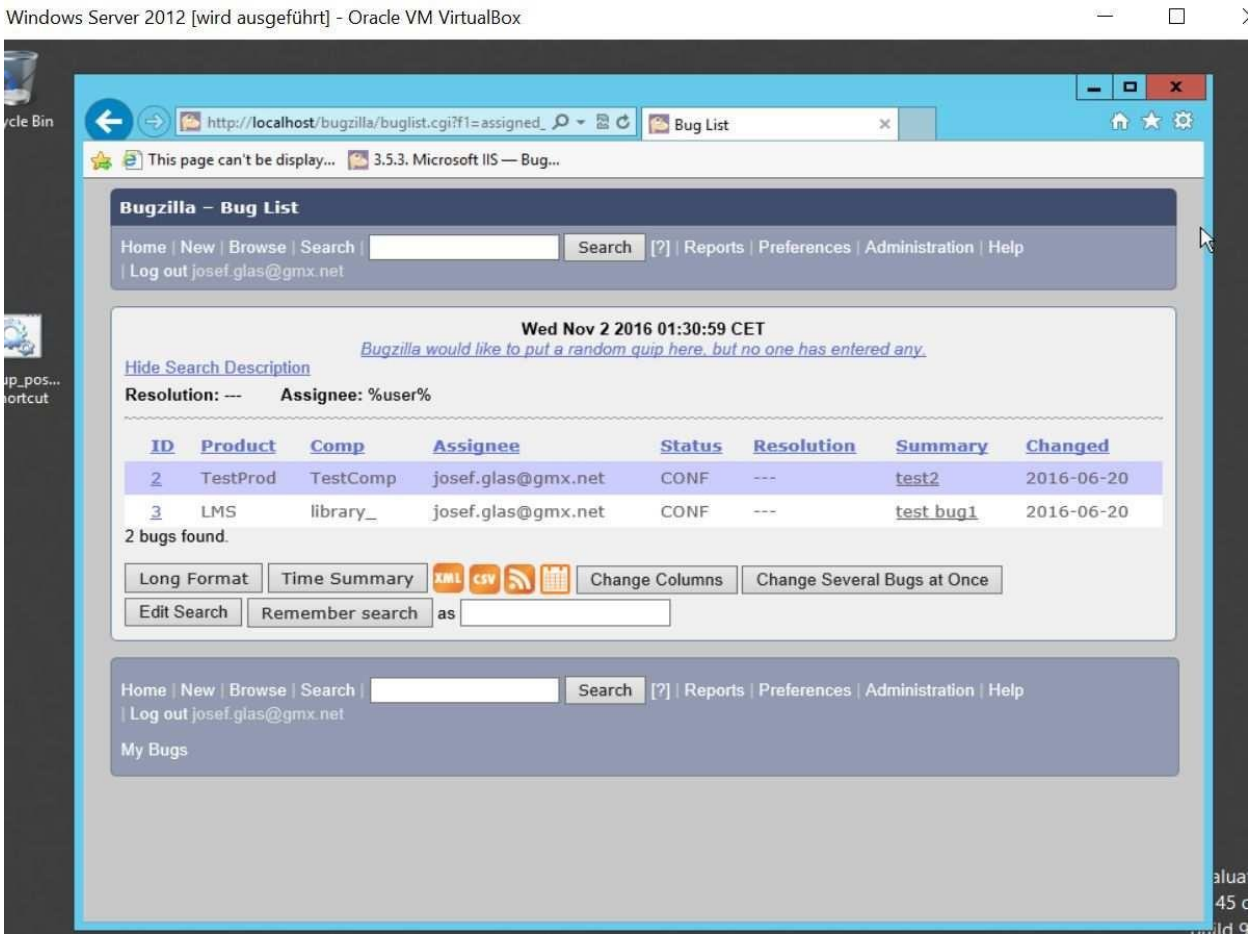

Figure 9: BugZilla: the module which supports [5]

- initiation of corrective actions

Also supported Mantis, Redmine, JIRA, Trac

In addition to Squash TM a toolbox is offered, called Squash TA (Test Automation).

"... is dedicated to the industrialization of automated tests execution. Squash TA natively interacts with various open source robots like Selenium. Squash TA is a management controller of automated tests of Web applications, web services and batches.

Squash TA's features are based on developments which were started in 2009 by Henix team. Squash TA has been made to answer to automation needs of Third Party software testing lead by Henix. These developments have been released to the Squash Community and constitute the base of Squash TA. [5]

Features [5]:

- The realization of robust automated tests

Squash TA can build automated tests less sensitive to the SUT (System Under Test) changes.

- The industrialization of your test executions

Your tests are reproducible at will through the possibilities to control the context and the execution environment.

- Universal : TA manages all your automated tests

Squash TA offers you to manage numerous types of automated tests : web applications, web services, batches. Moreover Squash TA may be deployed in agile environment and for ATDD methodology.

- Linkable with open source robots

Squash TA supports the main open source tools : Selenium I and II, Sahi, SoapUI...

3.1 Component Testing (unit test)

Unit testing is the practice of testing of the functions (methods) and analysis of the developer's code. Therefore developer tests some parts of his or her code after finishing it. This help developers to identify failures or errors in the code and improve the quality of the code. Also unit testing gives the developer a chance to see the features of the software in action. [9]

3.1.1 IntelliJ IDEA

IntelliJ IDEA features robust, fast, and flexible static code analysis. It detects compiler and runtime errors, suggests corrections and improvements before compile. This cross-platform IDE with own set of several hundred code inspections available for analyzing code on-the-fly in the editor. Below, in the Figure 12, there is a short example showing a careless mistake.

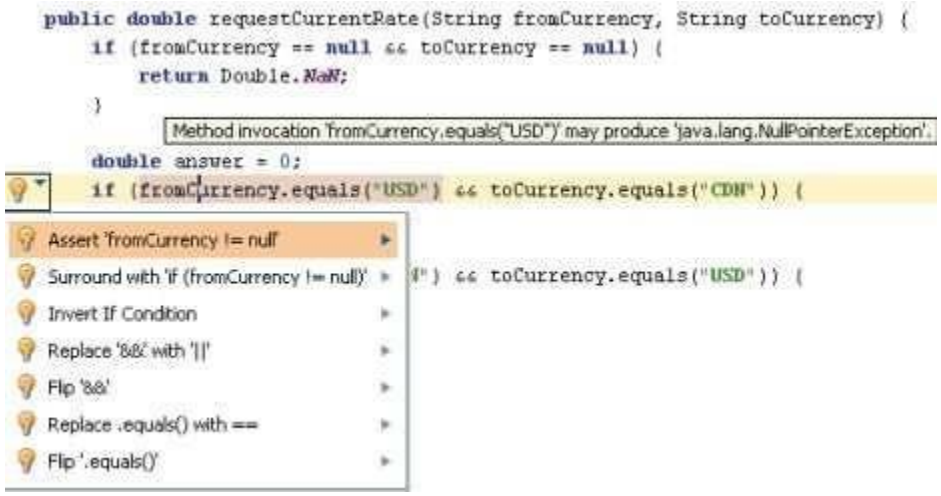

Figure 12: Finding probable bugs in IDEA

Here the first if-condition may lead to NullPointer exception being thrown in the second if, as not all situations are covered. At this point adding an assertion in order to avoid a NullPointer being thrown during the application runtime.
Figure 13 shows us exactly what we get from the intention action in IntelliJ IDEA build-in inspection code. [13]

```
public double requestCurrentRate(String fromCurrency, String toCurrency) {
    if (fromCurrency == null && toCurrency == null) {
        return Double.NaN;
    }

    double answer = 0;
    assert fromCurrency != null;
    if (fromCurrency.equals("USD") && toCurrency.equals("CDN")) {
        answer = rate;
    }
    if (fromCurrency.equals("CDN") && toCurrency.equals("USD")) {
        answer = 1 / rate;
    }
    return answer;
}
```

Figure 13: Fixed a compilation errors with inspection help of IDEA

3.1.2 JUnit

JUnit is a small and powerful open source Java framework for writing and running automated unit tests. The libraries for JUnit are shipped with IntelliJ IDEA and they can be easily added to

the project with the necessary library to the classpath automatically. With JUnit it is easier to write code faster. It increases the quality of code and it is less complex. All tests could be ordered in test suites and test cases and it saves much time. Since the tests are programmed directly in Java, testing with JUnit is as easy as compiling. The test cases are self-monitoring and therefore repeatable. In the Figure 14 below show the example of test fixture in which the purpose is to verify that a test is running in a fixed environment. [10]

```
import junit.framework.*;

public class JavaTest extends TestCase {
   protected int value1, value2;

   // assigning the values
   protected void setUp(){
      value1 = 3;
      value2 = 3;
   }

   // test method to add two values
   public void testAdd(){
      double result = value1 + value2;
      assertTrue(result == 6);
   }
}
```

Figure 14: fixed environment test with JUnit

3.1.3 DbUnit

Also for testing the units of the database layer, could be used DbUnit that is a JUnit extension. This extension is used for controlling a database dependency within applications by allowing developers to manage the state of a database throughout a test. The developers write code in a SQL language to extract the data. DbUnit has the ability to export and import database data to and from XML datasets. The main benefit of DbUnit is, that the framework drops the database and creates a new one with test data before every execution of a testcase. The framework comes with own declared methods to compare whole datasets with each other.

3.1.4 GroboUtils

When developer run a test case he gets only the result of the test Pass/Fail, but he doesn't know if he tested all written code. For this reason, there is GroboUtils that include code coverage tool, automated documentation and multi-threaded tests. For using code coverage first need to add *.jar file to the directory and change Ant build.xml file in the project with new reference and location of GroboCoverage.

Code coverage tool works by inserting "probes" into the post-compiled class files. These probes call out to a logger, telling it which class, method index and probe index was invoked. During the post-compilation of the class files, a set of data files get generated, which record what a method index and probe index translate into with regards to the source file.

In the Figure 15 below, there is a short example to show how to run the test with code coverage enabled using JUnit task.

```
<junit printsummary="yes" fork="yes" dir="${test-output.dir}">
<classpath>
        <pathelement location="coverage/classes" />
        <pathelement location="${classes.dir}" />
        <pathelement location="${test-classes.dir}" />
        <path refid="classpath.base" />
        <pathelement location="GroboCodeCoverage-1.1.0-runtime.jar" />
    </classpath>
<formatter type="xml" usefile="yes" />
    <batchtest todir="${test-output.dir}">
      <fileset dir="${test-classes.dir}">
        <include name="*Test.class" />
      </fileset>
    </batchtest>
  </junit>
```

Figure 15: code coverage test with using GroboUtils tool under JUnit

Here, to make priority in class loader, we adding coverage-enabled classes before everything else and we also added the *runtime.jar. The code coverage instrument told the logger running these tests and this will put all the coverage log information into the coverage/logs directory. [12] The reporting phase combines the post-compilation data files with the log data from the loggers to generate XML reports, and from that HTML reports using XSL-T.

3.1.5 jMock

Units of the source code is simple a classes of a programming language and describing the objects and behavior of the objects that are created from the class. In other words unit could be depends on another units. So if the developer want to test any unit he needs to know that this unit could have dependencies to other units and that mean that he needs to initialize the other affinity units too. Mock objects help a developer to test the interactions between the objects in the programs. For this purpose it easier to use an open source library JMock that give an opportunity to makes it quick and easy to define mock objects, precisely specify the interactions between objects and it is plugs into JUnit test framework, so it is easy to extend.
In Figure 16 show how to import jMock classes, define test fixture class and create a "Mockery" that represents the context in which the Publisher exists. The context mocks out the objects that the Publisher collaborates with and checks that they are used correctly during the test. [11]

```
import org.jmock.Mockery;
import org.jmock.Expectations;

public class PublisherTest extends TestCase {
    Mockery context = new Mockery();

    public void testOneSubscriberReceivesAMessage() {
        // set up
    final Subscriber subscriber = context.mock(Subscriber.class);

        Publisher publisher = new Publisher();
        publisher.add(subscriber);

        final String message = "message";

        // expectations
        context.checking(new Expectations() {{
            oneOf (subscriber).receive(message);
        }});

        // execute
        publisher.publish(message);

        // verify
        context.assertIsSatisfied();
    }
}
```

Figure 16: Import jMock classes

3.2 Integration testing

Integration testing is a kind of testing which the software modules are combined and will be stated as a group, this type of testing sometimes called grey-box tasting which means the tester partially knows about internal structure and algorithm of the software. Usually the integration testing starts after unit testing, after all components of the project have already been tested. The purpose of integration testing is to verify functional, performance, and reliability of the software. Testing of the integrated modules is designed to find latent defects as well as interface and database defects. [9]
The integration test depends on integration approach and in our project we use a Top-down approach which gives us a possibility first to test top modules in the hierarchy and then all extensions that are simulated with Mock objects. This type of integration approach could be time-consuming, but the benefit is that faults of the program could be been found easier and this testing could be performed with implementation.

TestNG is a Java testing framework inspired by JUnit and it is compatible with Eclipse, IDEA, Selenium, Maven. The test process could be simplified with using many advantages of TestNG. It uses annotation to provide a great way to test code in different ways: unit, regression, functional, integration and many others types of testing. It also allows to divide test methods in complex groupings, it means that not only methods can belong to the groups but also a group or the set of groups contain to other groups(MetaGroups) as it can be seen below in Figure 17. This gives a maximum suppleness in a meaning of how to partition the tests. This powerful framework generates its own reports in XML or HTML formats. [16]

```
@Test(groups = { "checkin-test" }) //new Metagroup
      public class All {
      @Test(groups = { "func-test" )
      public void method1() { ... }
      public void method2() { ... }
}
```

Figure 17: Adding a groups and methods in new MetaGroups

Continuous integration
Continuous Integration is a development practice that integrate the code to the entire team regularly in a shared repository to verify that it works all together. [18] When a developer commits his work, the whole system will be compiled and transferred to the test environment, where unit-tests, integration tests, system tests run automatically and this helps to detect integration error as soon as it possible.

GitLab CI is a open-source continuous integration service included with GitLab and after performing the tests in any programming language, the developers get immediate feedback if there is any problem. GitLab CI improves the quality of the product, documents all activities like build test and deploy. Errors in the code can be found more easily and be excluded in time. [17]

3.3 System Testing

Major focus at this level is the complete system. After component testing and integration all test we will focus on a consistent and integrated system.

3.3.1 Functional Testing

Functional testing is a kind of black-box testing, it means software or application will be tested under the condition that knowledge of the application's code/internal structure and programming knowledge in general is not required. The goal of functional testing is to verify that the application is behaving the way it was designed to. One of the best tools to test the functionality of web applications is Selenium. We assume, that all functions can be triggered using the GUI. See test cases in [1]

"Selenium is a web testing tool which uses simple scripts to run tests directly within a browser. In simple terms, "it automates browsers". It is a portable software testing framework for web applications that provides a record/playback tool for authoring tests without learning a test scripting language by using Selenium IDE". [15]

For a concrete example see chapter 4

3.3.2 Performance Testing

Performance testing is a non-functional testing which determine the system parameters in terms of scalability, stability and some others quality attributes. For normal behavior of the program in our project, we will use load and stress testing.

Load testing is one of the important and in the same time the simplest form of performance testing is a load testing. The load testing is performed to determine a system's behavior under normal and at peak conditions. It helps to identify the highest operating capacity of an application as well as any slow downs in the application and on the hardware side, determine which element is causing reduction.

Stress testing involves testing beyond normal operational capacity. Normally this kind of test is done to determine the system's robustness in terms of extreme load for example multi-user testing. It is performed to identify the defects in an application when multiple users login to the application and it also demonstrates the system's reliability by establishing that the various processes do not have a negative influence on each other. This test also helps in finding, analysing and measuring the complication in system parameters such as response time, throughput, locks/dead locks or any other issues associated with concurrency.

In our case, performance testing must be ensured that the system provides acceptable response times not more than 6 seconds on the server side. For this purpose, we will use Apache JMeter to test performance both on static and dynamic resources. By building a test plan is composed of a sequence of test components that determine how the load or stress test will be simulated. The output performance data could be done in several ways, including CSV and XML files, and graph.

From the figure 19 you can see how Apache JMeter perform performance testing.

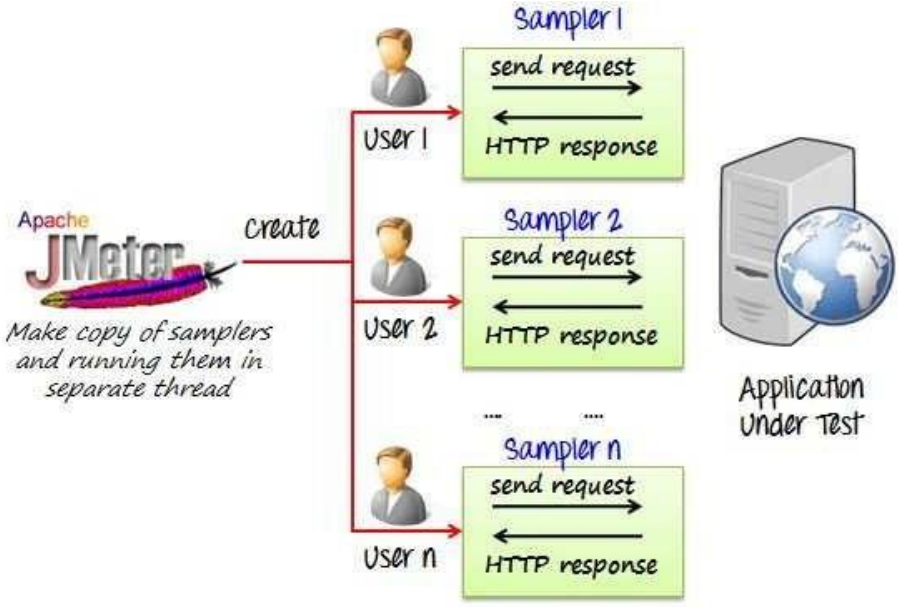

Figure 19: Heavy load simulating with JMeter [19]

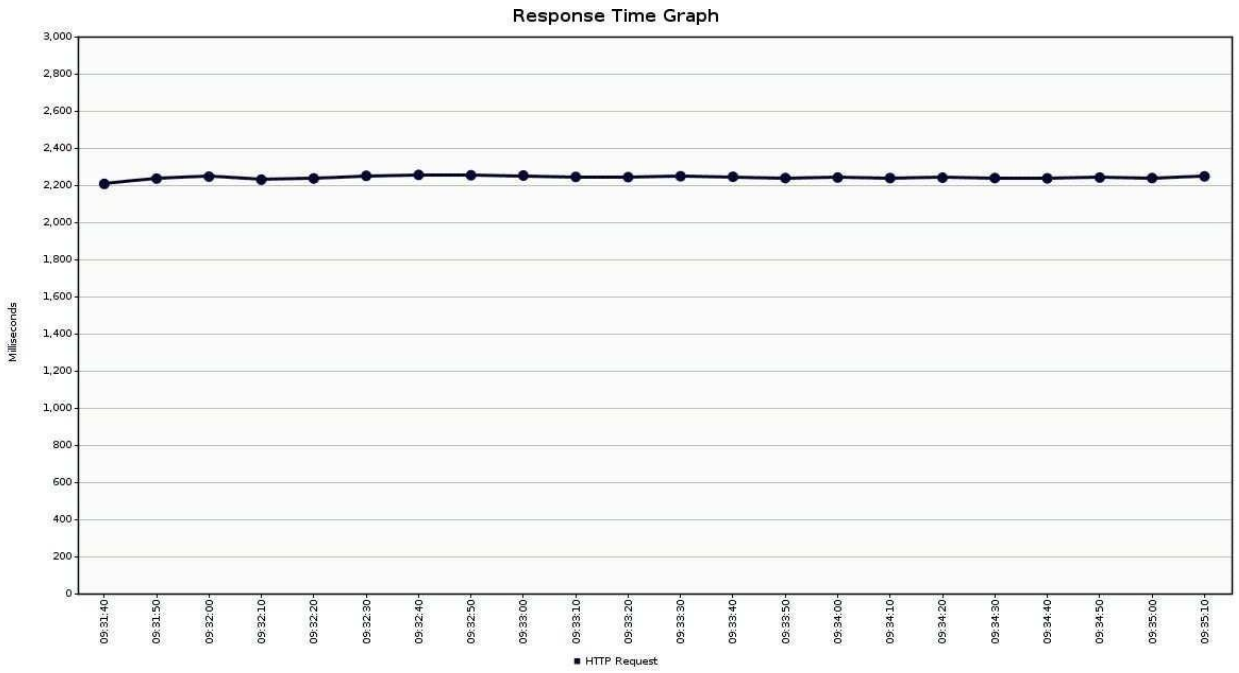

Figure 20: Response time graph from Apache JMeter

3.3.3 Security Testing

It is one of the most important technical challenges and as the customer belongs to an governmental organisation it become very important to produce secure software as security is a key limiting factor in deploying information technology in authority organizations. Security testing can help to build a secure application.
As security testing encompasses a wide area it become hard to give a definition of this testing. In general, it is a process of determining to what extent a system as a whole secures confidentiality, integrity and availability of information. Confidentiality is the prevention of intentional or unintentional unauthorized disclosure of content. The concept of availability ensures that information is accessible. Integrity concerns guarantee that information protected from being modified by unauthorized parties and that the information provided by the system is correct. [7]

In our case we need to make tests that are conducted to prevent any unauthorized access to the software code and modification of the data. In first discovery stage the purpose is to identify systems within scope and the services in use. Discovery phase is vulnerability analysis, in which the applicable vulnerability classes that match the interface are identified.
After the discovery stage, the goal of vulnerability scan is to find for known security issues by using automated tools and plugins like Black-box vulnerability scanners and match conditions with known vulnerabilities.

Also, for our project, we will use OWASP Zed Attack Proxy (ZAP), easy and open source free tool that provides automated scanners for OWASP top 10 vulnerabilities as well as a set of tools that allow to make professional penetration testing of security vulnerability in web applications.
Security testing ends with the security review stage. This stage does not apply any of approaches like discovery, vulnerability scan, vulnerability assessment, security assessment, penetration test or security audit. This is a confirmation that external or internal security standards are applied to system or product.

Remark: Security testing shall be performed also on component testing level
Narrative: A numerous numbers of security checks and vulnerabilities can be automated and found in the code review with using open source utility OWASP Dependency Check Gradle Plugin which could be easy integrated in IntelliJ IDEA project. Features of the OWASP Dependency Check can be run in a continuous integration to determine if there are new vulnerabilities discovered based on the addition of a new dependency, or the discovery of a new vulnerability in an existing dependency.
Of course, this plugin also provides numerous security inspections. These can inform a developer of potential security problems in the code. For example, in Figure 21 this code executes a dynamically generated SQL string, which might be susceptible to SQL Injection. [14]

```
private Customer getCustomerByIdFromDatabase(int customerId) {
    Customer customer = null;
    try (Connection connection = database.getConnection();
         Statement statement = connection.createStatement()) {

        try (ResultSet rs = statement.executeQuery("SELECT * FROM Customers WHERE id = " + customerId)) {
            rs.next();
            customer = new Customer(
                    customerId,
                    rs.getString("first"),
                    rs.getString("last")
            );
        }

    } catch (SQLException e) {
        doDatabaseErrorHandling(e);
    }
    return customer;
}
```

Call to 'Statement.executeQuery()' with non-constant argument

Figure 21: SQL potential vulnerability

3.4 Acceptance Testing

Before starting this phase of testing, according to acceptance criteria specified in project description in first two releases 9 of 10 test cases should be completed successfully and the pass rate 90% must be achieved before the software will be accepted for starting acceptance testing. Acceptance testing is usually a final phase of functional testing and it is done by main stakeholders of the project like users or customers. The aim of the acceptance testing is to check if all project requirements have been satisfied.

There are different types of acceptance testing. Most common is **user acceptance testing.** The general focus is on the usability and functionality of the system, hereby validating the fitness-for-use of the system by the user or customer.
This type is foreseen in the current planning, maybe we add also **operational acceptance testing.** This is performed with the aim to assure that the new system will be integrated in the existing production environment of the customer. The focus is on the operational issues like backup and restore procedure, restart capabilities, regular operational housekeeping tasks like database reorganization and so on.

We ignore **contract and regulation acceptance testing**, a system is performed against acceptance criteria as documented in a contract (legal and safety standards, norms and regulation), before the system is accepted.

4. Advantages and limitations, possible interfacing integration

Integration into IntelliJ IDEA

The libraries for JUnit and TestNG are shipped with IntelliJ IDEA, but are not included in the classpath of the project. So, when a test class is created, the references to the TestCase class or test annotations are not resolved. Therefore we should add it to the classpath of the project, we can use maven or we can directly add jar file to the classpath

Maven example:

```
<dependency> <groupId>junit</groupId> <artifactId>junit</artifactId>
<version>4.12</version> <scope>test</scope> </dependency>
```

And direct classpath example :

- Go to File -> Project Structure
- Now select Modules and then "Dependencies" tab
- Click the "+" icon and select "Library"
- Click "New jars or directories" and select the jar file
- And finally click "Add Selected"

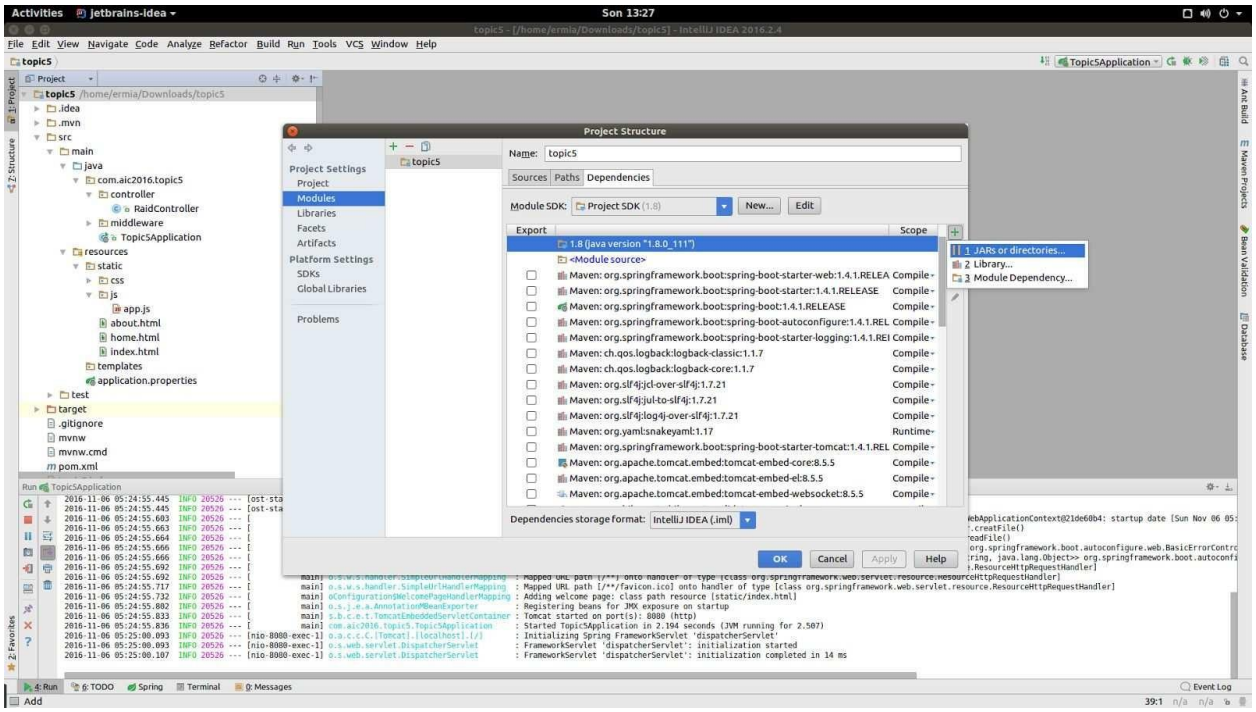

Figure 22: Screenshot IntelliJ - add dependency

Integration into Squash TM and Squash TA - Overview

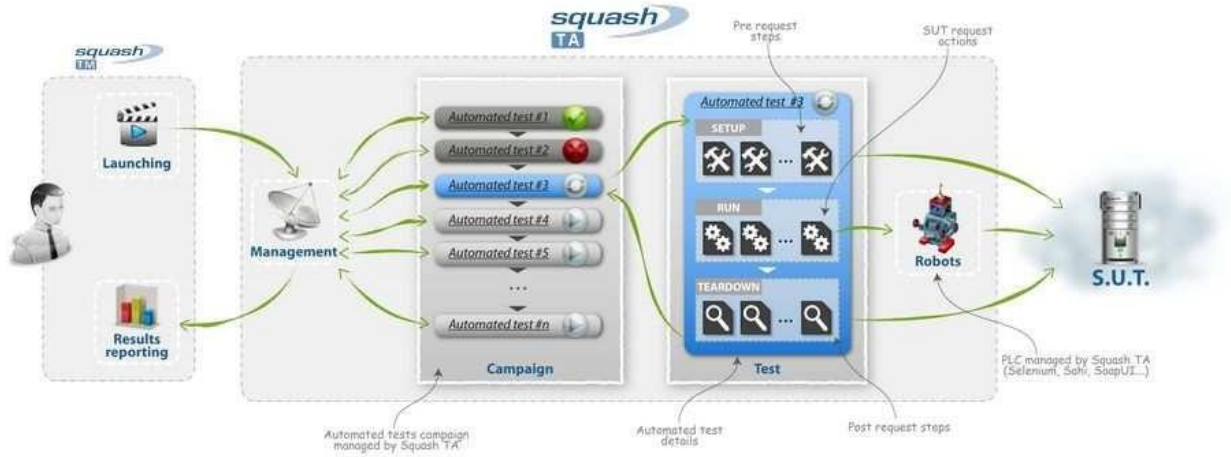

Figure 23: Squash TM and Squash TA integration

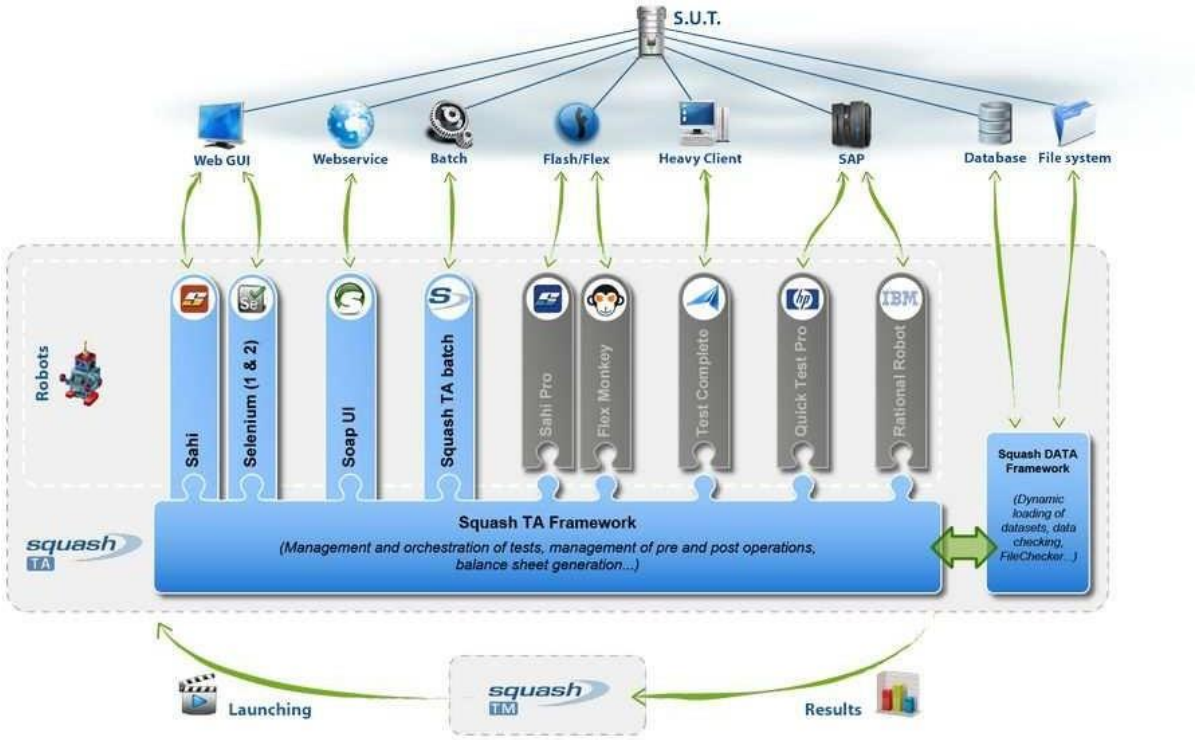

Figure 24: How Squash TM acts as Framework and extends existing Robots

Integration of Squash TA and Selenium - a practical example

To test our project GUI functionality we should install firefox browser and Selenium IDE, after that we should capture user behavior with help of Selenium IDE then export the test case as Junit (webdriver).

We can then create folder "tests" under Squash TA project in Intellij and then add following file

MyprojectSeleniumTest.ta then open this file and under Test section modify it as follows

TEST :
 # EXECUTE_SELENIUM2 selenium WITH MAIN CLASS MyprojectSeleniumTest

Then execute the Squash-TA testcase in IntelliJ (or Eclipse). After the execution the result can be inserted to the database; assuming proper setup of database.

As you can see from figure below, this is an example from Selenium IDE output as JUnit code.

```
import  java.util.regex.Pattern;
import java.util.concurrent.TimeUnit;
import org.junit.*;
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.openqa.selenium.*;
import   org.openqa.selenium.firefox.FirefoxDriver;
import  org.openqa.selenium.firefox.FirefoxBinary;
import  org.openqa.selenium.firefox.FirefoxProfile;
import java.io.File;
import org.openqa.selenium.support.ui.Select;

public class PetStoreSeleniumTest {
  private FirefoxBinary binary = new FirefoxBinary(new File("C:/Squash-TA/Firefox12/firefox.exe"));
  private FirefoxProfile profile = new FirefoxProfile();
  private WebDriver driver;
  private String baseUrl;
  private boolean acceptNextAlert = true;
  private StringBuffer verificationErrors = new StringBuffer();
  @Before
  public void setUp() throws Exception {
    driver = new FirefoxDriver(binary,profile);
            baseUrl = "http://localhost:8080/";
            driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);  }
  @Test
  public void testPetStoreSelenium() throws Exception {
            driver.get(baseUrl);
            driver.findElement(By.linkText("Enter the Store")).click();
            driver.findElement(By.xpath("//tr[5]/td/a/img")).click();
            driver.findElement(By.xpath("//tr[3]/td/b/a/font")).click();
            driver.findElement(By.xpath("//tr[3]/td[5]/a/img")).click();
            driver.findElement(By.xpath("//td[2]/center/a/img")).click();
            driver.findElement(By.xpath("//center[2]/a/img")).click();
            driver.findElement(By.name("username")).clear();
            driver.findElement(By.name("username")).sendKeys("j2ee");
            driver.findElement(By.name("password")).clear();
            driver.findElement(By.name("password")).sendKeys("j2ee");
            driver.findElement(By.name("update")).click();
            driver.findElement(By.cssSelector("p > input[type=\"image\"]")).click();
            driver.findElement(By.xpath("//center[3]/a/img")).click();  }
  @After
  public void tearDown() throws Exception {
            driver.quit();
            String verificationErrorString = verificationErrors.toString();
            if (!"".equals(verificationErrorString)) {
            fail(verificationErrorString);  }
  }
  private boolean isElementPresent(By by) {
            try {
            driver.findElement(by);
            return true;                                    }
            catch (NoSuchElementException e) {
            return false;                                   }
  }
}
```

Figure 25: Selenium JUnit example

5. Cost / benefit criteria

This chapter provides cost and licensing information for proposed test tooling.
Here we focus only on cash out positions related to the licensing of the software products. The costs needed für consultancy, training, implementation or customization effort is not considered. It is assumed that this know-how for the project comes with the testing team.

We do not analyse the benefits of the proposed tooling. We assume there will be further development of the SUT which will result in many succeeding versions. Hence, the introduced tooling and the possibility for automated regression testing can easily show a positive business case.

List of testing tools in alphabetic order:

Apache Ant
Software tool for automating software build processes. It is covered by the Apache License 2.0.

BugZilla
Mozilla Public License

GitLab CI
It is a part of GitLab. Can be used for free on GitLab private repositories under the MIT License.

GroboUtils
Open source, aims to expand for the testing possibilities of Java

IntelliJ IDEA - Community edition
It is open source and available on free of charge and it is covered by the Apache License 2.0.
We can say that this edition is enough for using and add extension like JUnit (DBunit, JMock, TestNG, OWASP Dependency Check) that are also open source and free for using.

MS Project
We assume this is the preferred planning tool of the customer. Licenses (if needed) are provided by the customer.

Open Office
Apache OpenOffice releases are made available under the Apache License 2.0.

OWASP Zed Attack Proxy (ZAP)
ZAP is open source web application security under Apache License 2.0.

Selenium
is a portable software testing framework for web applications under Apache License 2.0.

Squash
Squash is an open source software, distributed under LGPL v3 license. The sources are available on our Bitbucket repository.

6. Appendix

In the appendix we present secondary documentation which have been created in the course of this exercise.

6.1 Installation logs

	Squash TM	OS	DB	plug-ins	JVM	others
Ermia	1.13.2.	Debian	MySQL 5.x	Jira report.books.distribution	1.7	Performance Testing, TA installation, screenshot
Josef	1.13.2.	Windows Server 2012	Postgre SQL 9.x	Bugzilla report.books.distribution	1.7	Fundamental test process

Additional software: Squash TA (test automation), See www.shashtest.org

VirtualBox (Windows) - Josef

1. Install OS
Windows Server 2012 Datacenter Evaluation (login: Administrator/Sqashtest.org)

2. Install Java 1.8.0_91

3. Install Database
3.1 PostgreSQL 9.x (login: postgres/postgres)
3.2 run postgresql-full-install-version-1.13.3.RELEASE.sql

4. Modify startup script:
C:\squash-tm-1.13.2.RELEASE\squash-tm\bin\startup_postgres.bat

5. Run startup script

6. Connect to squash-tm
http://localhost:8080/squash/ (login: admin/admin)

7. Install Plugin Bugtracker BugZilla.zip v.1.0.0

8. Install BugZilla 5.0.3 (www.bugzilla.org)
- ActivePerl 5.24.0 for Windows 64-bit
- required Perl modules installed with ppm
- setup IIS
- configure postgreSQL -> create user "bugs/bugs"
- perl checksetup.pl -> followed instructions
- BugZilla Administrator -> josef.glas@gmx.net PW=josefx

BugZilla Test:
PS C:\bugzilla-5.0.3> perl testserver.pl http:/./localhost/bugzilla
sysread() is deprecated on :utf8 handles at C:/Perl64/site/lib/File/Slurp.pm line 225.
TEST-OK Got padlock picture.
TEST-OK Webserver is executing CGIs via mod_cgi.
TEST-OK Webserver is preventing fetch of http://localhost/bugzilla/localconfig.

TEST-OK GD library generated a good PNG image.
TEST-OK Chart library generated a good PNG image.
TEST-FAILED Template::Plugin::GD is not installed.

SquashTA
- prerequisites: JDK, Firefox

SquashTA server
- Apache Maven
- Sahi
- Jenkins
- Apache Tomcat

6.2 Generic types of testing [4] relevant on each test level:

A. Functional testing (based on functional requirements)

B. Non-functional testing
- Load test
- Performance test
- Volume test
- Stress test
- Testing of security
- Robustness test
- Testing of compatibility and data conversion
- Testing of different configurations of the system
- Usability test
- Checking of the documentation
- Checking maintainability

C. Testing of software structure

D. Testing related to changes and Regression testing

7. References

[1] *Library Management System*, SQM-Project-Description-SS2015.pdf, INSO, 2016

[2] Breiteneder R.: *Fundamental Test Process*, SQM_SS16_VO_1.pdf, INSO, 2016

[3] Breiteneder R.: *Vorbesprechung*, SQM_SS16_Vorbesprechung.pdf, INSO, 2016

[4] Spiller A., Linz T., Schaefer H.: *Software Testing Foundations*: A Study Guide for the Certified Tester Exam, Rocky Nock Computing, 2014

[5] Open source project, test management and test automation, http://www.squashtest.org

[6] Overview of testing tools https://www.testtoolreview.de/en

[7] Derk-Jan de Grood, Collis B.V.: *TestGoal*: Result-Driven Testing, 2008

[8] Types of Software Testing http://www.testingexcellence.com

[9] Practical Guide to Software Quality Management John W. Horch

[10] JUnit - Test Framework https://www.tutorialspoint.com

[11] Open source library JMock, the overwiew http://www.jmock.org

[12] GroboUtils Home Page http://groboutils.sourceforge.net

[13] JetBrains, creator of the best Java IDE - IntelliJ IDEA https://www.jetbrains.com

[14] What to look for in a Code Review: Security https://blog.jetbrains.com

[15] Automating Functional Testing using Selenium http://www.tatvasoft.com/

[16] TestNG. Java testing. Next generetion http://testng.org/

[17] GitLab Continuous Integration https://www.gitlab.com

[18] S. Daya, N. Van Duy, K. Eati, C. M. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins, S. Narain, R. Vennam.; Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach. IBM Redbooks, 2016

[19] Performance Testing using Jmeter http://www.guru99.com/