SOTE Assessment

Y3864454

Table of contents

1 Test Plan

1.1 Purpose

The main objective of this document is to test the functionality of the JOrtho application at a range of testing levels, including unit, integration and system level. This Test Plan document supports the following objectives:

- Identify existing information about the application that need to be tested.
- Elicit the explicit and implicit test requirements.
- Define the overall strategy for creating test cases.
- List the acceptability criteria for the software.

1.2 Project Overview

JOrtho is a spelling-checking library coded in Java which provides potential users (e. g. application programmers) with the ability to check the correctness of a particular word. It supports different languages and is based on Wiktionary project allowing users to define their own dictionaries for words. It has a GUI interface which highlights the potentially wrongly spelled word and offers a context menu with suggestions for a correct form of the word [1].

The functionality of this application can be used by application programmers who can integrate it into the application they develop.

1.3 Scope

This Test Plan describes the functional tests that will be conducted on the JOrtho application through white-box testing. The listing below defines the features that have been identified as targets for testing. This list represents what will be tested followed by the features what will not be tested.

1.3.1 Features to be tested

1. Verify access to the dictionary file, i.e. the application can successfully load the dictionary file.
2. Verify the application highlights the wrongly spelled word.
3. Verify a user can add a new word to his own dictionary.
4. Verify a user can view a list of alternatives for the wrongly spelled word.
5. Verify a user can change a wrongly spelled word by the one from the list of alternatives;
6. Verify an application detects invalid words which could be: 1) wrongly spelled words; 2) words containing digits; 3) the first words of the sentences start with lower case

1.3.2 Features not to be tested

1. User interface testing, e.g. sample screens conform to GUI standards, ease of navigation through a sample set of screens etc.
2. Performance testing, e.g. memory usage.
3. Configuration testing, e.g. the usage of the application on different operational systems.
4. A particular functionality, i.e. functional testing: the ability to change language, the ability to add new language, the application ability to show informative warning/errors dialogs if an error occurs etc.

1.4 Strategy for determining expected behaviour

As we do not have the explicit requirements specifications, we are going to discover them by two approaches defined by C. Kaner [2]. The first is inference, i.e. extrapolating application requirements from the behaviour of similar libraries such as Hunspell [3], JSpell [4], SpellCheck and also from likely user assumptions (e.g. the system should highlight the wrong spelled word). By testing the functionality of the beforementioned applications which interfaces are shown in Figure 1 (SpellChecker library) we know the behaviour of spell-checking libraries, i.e. what functionality we should expect from the JOrtho software:

highlighting invalid words, suggesting a list of alternatives for invalid words, changing all invalid words in the text, detecting digits in invalid words and other features.

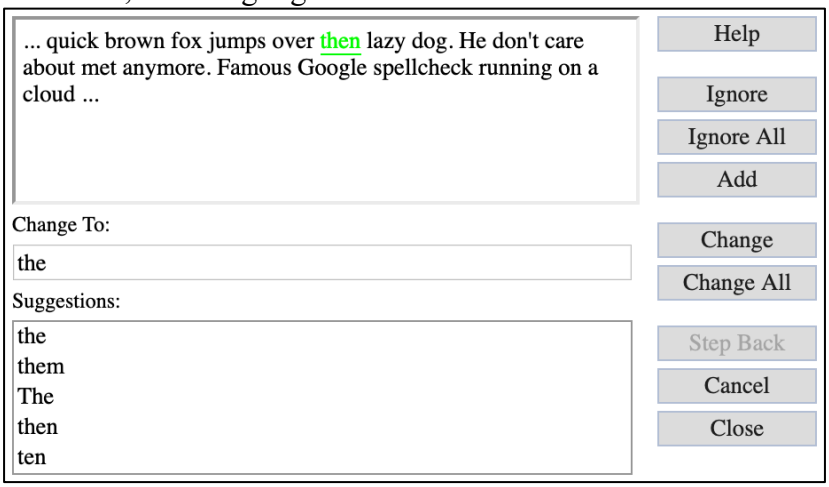

Figure 1 SpellChecker library

The second way is reference, i.e. discovering requirements from the implicit as well as explicit specifications, e.g. uses implied by JOrtho's demo programs and documentation. Some clearly stated requirements are defined in the JOrtho documentation [1]. We are not going to duplicate here the requirements stated in the JOrtho documentation to save the space.

Thus, by testing the applications with the similar functionality, we elicit implicit requirements – the expected behavior of the tested software. Then, by having looked at the application's documentation, we have known the explicit requirements, i.e. what the system should do, its functionality and what its purpose. Hence, we have achieved the main objectives during requirements analysis: requirements are testable, complete and correct [5].

1.5 Strategy for building the test cases

The test strategy described in this section provides with the recommended approach to the testing of the JOrtho library. We are going to undertake requirements-based testing in accordance with the requirements which have been elicited before. The test strategy combines different types of functional testing beginning with unit testing and ending by system testing (Table 1). We use functional testing approach as our primary objective is to test the functionality of the JOrtho library. As we are going to test code by executing it, we will use dynamic testing. What is more, as we exploit knowledge of the system to design tests for it and know all source code, we will use white-box testing technique.

1.5.1 Levels of Testing

Table 1 The testing types

Testing Level	Test objective	Technique	Example	Minimum Exit Criteria
Unit testing	to verify that each unit of the application performs as designed	Execute a particular function/set of functions providing valid and invalid (e.g. words with digits/symbols) data to verify: • the expected results occur when	Test a particular low-level function: if a user adds a new word to his own dictionary, the dictionary array must not be empty. So, we need to test whether the array empty with *assertTrue()* helper function.	Unit Test Code Coverage goal <xx %>
System testing	to evaluate that the system's behavior conforms to the defined requirements		Evaluate the application performance against to expected behavior: e.g. the application refreshes the highlighting if the dictionary was modified.	All test's scenarios passed No Critical / High defects are open

Integration testing	to verify that communication amongst different modules is correct	correct data is provided; • the appropriate warning/error dialogs are displayed when invalid data is provided;	The red highlighting is appeared if the word is spelled incorrectly. So, we need to test the communication amongst separate modules: *RedZigZagPainter.class*, *CheckerListener.class* and others	All documented acceptance criteria met.
Module testing	to evaluate the concrete module (class) behaves correctly		Test behavior of particular class, e.g. the *Dictionary.class* is successfully initializes after the application has been launched	All test's scenarios passed

1.5.2 Tools

Table 2 shows the tools which are used during all cycles of software testing.

Table 3 The tools

Tool Function/Purpose	Tool
Gathering requirements	Microsoft Office Word 2016, Google Chrome Browser, Eclipse IDE
Performing testing	Eclipse IDE: JUnit tests, Microsoft Office Word 2016
Analyzing the results, measuring code coverage	Microsoft Office Word 2016, Eclipse IDE: EclEmma

A windows/linux environment with Eclipse IDE version 4.10, as well as Google Chrome 32.0 and later should be available to each tester.

1.6 The acceptability criteria

In order to verify the application has an acceptable quality, we are going to define acceptability criteria for the JOrtho library which are listed below:

- A user should be able to add new words to his own dictionary.
- The application correctly loads the dictionary from the dictionary file.
- A user should be able to view the list of alternatives for an invalid word.
- A user should be able to know that a particular word is spelled incorrectly: spelling error, capitalization error, digits in the word.
- A user should be able to change the wrongly spelled word by clicking right mouse button.
- A red highlighting should disappear once a user: 1) has added the wrongly spelled word to his dictionary; 2) a user has changed the wrongly spelled word.
- A text in the text panel should be refreshed when a user makes changes.

2 Test Case Specifications

This chapter describes the test case specifications for 8 fully specified test cases at a range of testing levels, including unit, system, integration, module levels (Table 4). For all testes we assume that a user would use English language and check the spelling of English words. So, before using the application a user should change the language to English.

To structure tests and to execute them correctly, we are going to develop 5 different test suites which are shown in Figure 2. This approach makes the testing scalable, i.e. if we need to develop a new test, firstly, we need to define it into a particular group (module, unit, system etc.) and then just add it into the appropriate Java class. Thus, we keep the testing logic clear and understandable for other testers, i.e. the test are well-structured and easy to reproduce.

We are going to perform a testing by following the error seeding approach. First, we will provide correct input to see how the application behaves and then we will provide incorrect input to catch any potential bugs.

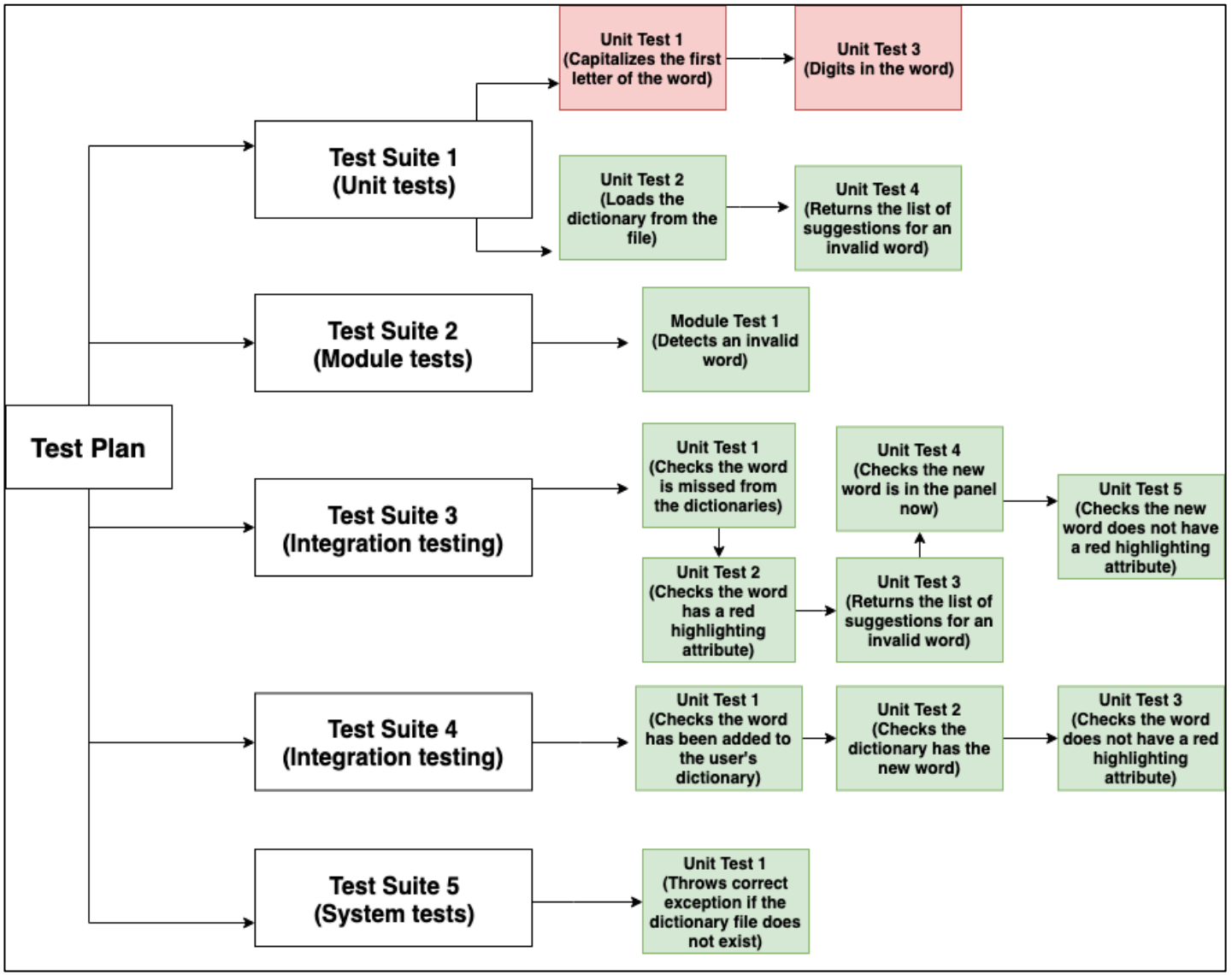

Figure 2 The description of test suites

Table 4 Test Cases

№	Test Type	Purpose	Stimulus	Input	Expected behaviour	Comment
1.	Unit test	check if the first character of the first word in the sentence is a lower letter	Constraint: 1.the first character of the first word in the sentence should be in lowercase. 2. the first word in the sentence should be correctly spelled 1. Launch the application	"**this** is a simppler textt with spellingg errors. **hello** how are y6ut?"	From a user perspective: The first word in the sentence has a red highlighting: *this, hello* From a tester perspective: *assertTrue()* statement should be successful	According to the the JOrtho library documentation, it should detect whether the first word in the sentence starts with a lower case and in this case it should be highlighted with a red line. The class *Utils.class* is responsible for this feature. The function *isFirstCapitalized(word)* should be tested.
2.	Unit test	to check the application loads the dictionary from the *dictionary_en.ortho* file successfully	Constraint: the file *dictionary_en.ortho* should exist in the folder *"JOrtho_1.0/bin/"* 1. Launch the application	-	From a user perspective: the text panel appears on the screen From a tester perspective: the dictionary array shouldn't be empty	In order to use the functionality of the JOrtho library, the most important thing is to load a dictionary from a file. The class *Dictionary.class* is responsible for this feature. The function *load (filename)* should be tested.
3.	Unit test	to test the application returns a list of suggestions if the word is not in the dictionary	1. click the right mouse button on a word with a red highlighting in order to view possible options; 2. from two available options move cursor on *"Suggestions"* option; 3. move cursor to the last option from the list *"List of suggestions"*	"this is a simppler **textt** with spellingg errors. hello how are y6ut?"	From a user perspective: a list of alternative words should appear on the screen: *text, texts, test etc.* From a tester perspective: an array of alternative words shouldn't be empty.	One of the most important functions is to change a wrongly spelled word by one of the suggested alternatives. So, a list of alternatives should be successfully generated. The class *Dictionary.class* is responsible for this feature. The function *searchSuggestions (word)* should be tested.
4.	Unit test	check if a word contains digits in the start of the word	Constraint: 1. the word should be correctly spelled 2. the word should start with digits 1. Launch the application	"this is a simppler textt with spellingg errors. hello how are **6you**?"	From a user perspective: the word with digits has a red highlighting From a tester perspective: *assertTrue()* statement should be successful	According to the the JOrtho library documentation, it should detect whether the word starts with digits and in this case it should be highlighted with a red line. The class *Utils.class* is responsible for this feature. The function *isIncludeNumbers (word)* should be tested.

5.	Module test	to verify that the application correctly detects a wrongly spelled word	1. Launch the application 2. Type a word with mistake	"this is a **simppler textt** with **spellingg** errors. hello how are **y6ut**?"	From a user perspective: the wrongly spelled words should be red highlighted From a tester perspective: a *Tokenizer.class* should firstly return the word *simppler, then textt, spellingg, y6ut*	According to the JOrtho library, a wrongly spelled words should be detected and highlighted with a red line. This is a Module test because we test the performance of the whole module – *Tokenizer.class* which is responsible for detecting invalid words in the text. The function *nextInvalidWord ()* should be tested.
6.	Integration test	to test the ability of a user to change a wrongly spelled word by one from the list of alternatives	1. Click the right mouse button on a word with a red highlighting in order to view possible options; 2. From two available options move cursor on *"Suggestions"* option; 3. After having look at all suggested words, click on the one which suits the best	"this is a simppler **textt** with spellingg errors. hello how are y6ut?"	From a user perspective: 1. the wrongly spelled word should be changed by the user's chosen word in the text panel, e.g. *textt* may be changed by the word *text* 2. the red highlighting should disappear immediately From a tester perspective: 1. a language dictionary array doesn't contain the wrongly spelled word; 2. a user's dictionary doesn't contain wrongly spelled word; 3. the wrongly spelled word has a highlighting attribute 4. an array of suggestions for this word contains minimum 15 entries; 5. text in the text panel has new word; 6. a new added word doesn't have a red highlighting attribute	This test case can be split into 5 tests: 1. a test which checks that dictionaries (user's and language's) don't contain this wrongly spelled word 2. a test which checks that the word has red highlighting 3. a test which ensures that a list of suggestions for this word has been generated correctly 4. a test which checks that selected by user's alternative word has changed old wrongly spelled word in the text panel 5. a test which checks that an alternative word doesn't have a red highlighting To implement that, we can create a specific class which is responsible for only testing this feature, e. g. *TestChangeWrongWord.class.* Thus, we have a test case that is doing integration testing.
7.	Integration test	to test the ability of a user to add a new word, which the application identifies as wrongly spelled	1. Click the right mouse button on a word with a red highlighting in order to view possible options;	"this is a simppler **textt** with spellingg errors. hello	From a user perspective: 1. the word should be successfully added to the user's dictionary, i.e. the word should appear in the Dictionary panel;	This test case can be split into 3 tests: 1. a test which ensures that a new word has been successfully added to the user's dictionary 2. a test which checks the word is then in the dictionary

		and highlights with a red line, to his own dictionary from a context menu by clicking the right mouse button. In other words, to check the link between Edit Words module, Suggested Words module and Dictionary module	2. From two available options move cursor on *"Suggestions"* option; 3. Move cursor to the last option from the list *"Add to the dictionary"* and click on it;	how are y6ut?"	2. the word's red highlighting should disappear immediately; From a tester perspective: 1. the user's dictionary array should not be empty; 2. the user's dictionary array should contain the new word; 3. the word should not have highlighting attribute	3. a test which verifies that that word doesn't have a red highlighting To implement that, we can create a specific class which is responsible for only testing this feature, e. g. *TestAddingNewWord.class.* Thus, we have a test case that is doing integration testing.
8.	System test	to verify the application throws *FileNotFoundException* if the dictionary file does not exist	Constraint: the file *dictionary_en.ortho* should be deleted from the folder *"JOrtho_1.0/bin/"* 1. Launch the application	-	From a user perspective: the application should show an error message with the error's description – e. g. file with the dictionary has not been found. From a tester perspective: *FileNotFoundException* should be caught	As the only requirement for a system testing is an application should be fully deployed, we can perform this type of testing in our case because we test an already deployed software. To test the application throws *FileNotFoundException*, an annotation *@Test (expected = java.io.FileNotFoundException.class)* should be used which allows to test the ability of the application to throw an exception if it cannot get access to the *dictionary_en.ortho* file.

3 Test Results

After having developed different types of test cases, which have been described in the previous section, they have been successfully implemented and applied for testing of the JOrtho library. To structure them Each test has been executed separately documenting the behavior of the software. Tests which are included in complex test cases (e.g. integration tests), were executed separately and then all together in order to test the communication amongst them, i.e. to perform integration testing. You can find the result of testing in the Table 5.

Table 5 Test Results

№	Test Type	Passed/Failed	Comments
1.	Unit test	Failed	If the sentence starts with the word which first letter is in lowercase, the word does not have a red highlighting. This is shown in Figure 4.
2.	Unit test	Passed	Meet the expected behavior
3.	Unit test	Passed	Meet the expected behavior
4.	Unit test	Failed	The invalid word *6you* which starts with digits does not have a red highlighting. This is shown in Figure 3.
5.	Module test	Passed	Meet the expected behavior
6.	Integration test	Passed	Meet the expected behavior
7.	Integration test	Passed	Meet the expected behavior
8.	System test	Passed	Meet the expected behavior

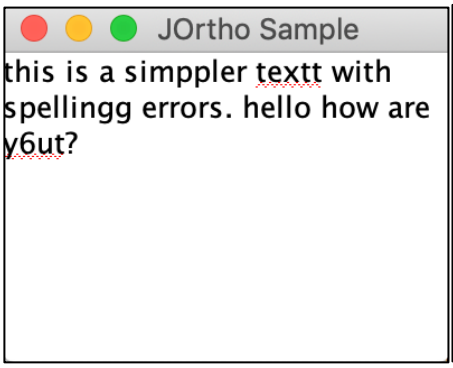

Figure 3 Failed Unit Test

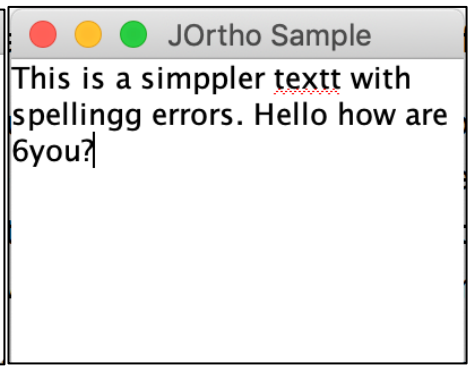

Figure 4 Failed Unit test

Overall, from 16 tests which have been carried out only two test failed with other 14 passed. The failed tests are shown in Figure 2 with a red background. The tests which failed are one of the most important features of the JOrtho library, so it a has major severity rating and should be fixed immediately.

4 Test Summary Report

4.1 A summary of the testing performed

In summary, 8 fully specified test cases were developed and 16 tests were implemented, including unit testing, module testing, integration testing and system testing. Additionally, test were divided into 5 test suites categories in order to increase the scalability of the testing. Each test in a particular test suite was executed separately as well as in conjunction with others. In the stage of the integration testing a testing of communication between particular functions and modules were performed. Overall, we have implemented 12 unit tests, 2 integration tests, 1 module test and 1 system test that is in total 16 tests on different levels – low and high-levels. It should be noted that some unit tests from those 16 implemented tests are parts of others high-level tests such as integration test. During the implementation stage 2 faults have been detected which significantly influences the functionality of the software.

4.2 A summary of the results

In this chapter we will describe the faults which were identified during the testing stage. The failures were detected on the low-level testing - unit testing. Only two fault were identified which is 13% from all tests which passed with 87% respectively.

Test Description: check if the first character of the first word in the sentence is a lower letter
Input: this is a simpler textt with spellingg errors. hello how are y6ou?;
Output: this is a simpler textt with spellingg errors. hello how are y6ou?;
Failure: the words **"**this" and "hello" should have a red highlighting;
Expected behaviour: this is a simpler textt with spellingg errors. hello how are y6ou?;
Severity rating: major as it directly decrease the functionality of the library

Test Description: check if a word contains digits in the start of the word**;**
Input: This is a simpler textt with spellingg errors. Hello how are 6you?;
Output: This is a simpler textt with spellingg errors. Hello how are 6you?;
Failure: the word **"**6you" should have a red highlighting;
Expected behaviour: This is a simpler textt with spellingg errors. Hello how are 6you?;
Severity rating: major as it directly decrease the functionality of the library

The summary pie chart which displays the results of the testing performed is shown in Figure 5.

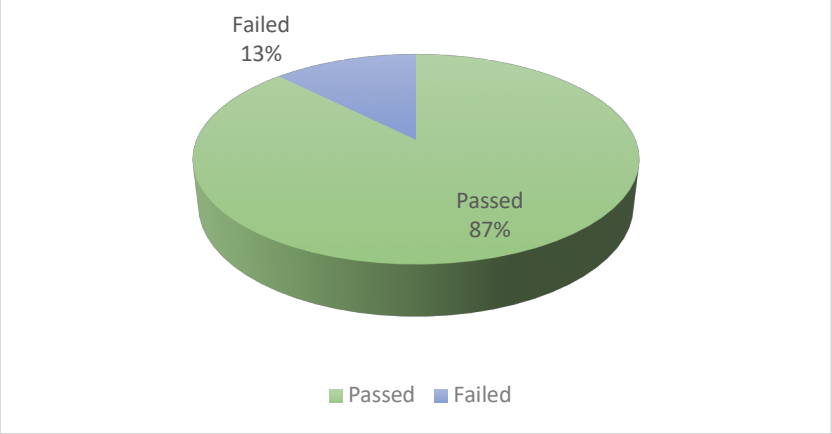

Figure 5 The result of testing

4.3 An evaluation of the quality of the testing performed

The code coverage of the software is 21% which is good in the given possibilities (the allowed number of test cases). In summary, the classes which have been covered are *Tokenizer.java* with 49.5% code coverage, *Dictionary.java* with 72,5%, *Utils. java* with 24.9%, *RedZigZagPainter.java* with 42.2.%, *Suggestions.java* with 54.7%, *LanguageBundle.java* with 78.7%.

Overall, the testing which has been performed on JOrtho library could be performed more efficiently if the following possibilities were available:

- People with similar competencies and expertise to the author of this paper work, i.e. the peer review or peer desk-check strategies. It would likely allow to find more software weaknesses;
- In terms of software: a lot of different frameworks make the testing process easier and faster, e.g. Arquillian [6] which allows to easily develop automated integration, functional and acceptance tests for Java-based software. Another powerful framework is Mockito [7] makes possible for developers to create and test mock objects in

automated unit tests for the purpose of Test-driven Development (TDD) or Behavior Driven Development (BDD).

4.3.1 The mutation testing approach

Additionally, to assess the quality of the testing performed we are going to undertake a mutation testing producing the mutation score, which indicates the quality of the input test set [8]. First, we will deliberately mutate, i.e. change, some parts of the code. Second, we will execute these mutant programs against the input test in order to assess the quality of the developed before test cases which should be resistance enough to fail changed code. The changes of the code which we are going to implement is the modifications of the expression statements and operators statements. For example, if the initial expression is

```
if( Character.isDigit( ch ) ) {return true;}
```

we will insert the symbol *"!"* in order to have a *"not"* statement. Having executing the test set against this mutant, the Unit test which checks whether the word starts with digits failed. It means that this test has been implemented correctly and is resistant to mutant programs, i.e. it is able to catch errors in the program.

However, inserting mutants manually is a time-consuming process. For this reason, a lot of automation tools are widely used. The most popular of them are Pitest [9], Stryker [10] and many others. We will use Pitest to asses the quality of the testing performed. The results of the Pitest analysis is shown in Figure 6. As we can see from the Figure 6, different types of mutant programs have been created, including *ConditionalsBoundaryMutator, IncrementsMutator, ReturnValsMutator, MathMutator* and *NegateConditionalsMutator*. Additionally, we can observe that all generated mutants have been killed which means that the implemented tests are mutation adequate. Thus, the **mutation score is 100%.**

```
12:48:45 PIT >> INFO : MINION : 12:48:45 PIT >> INFO : Found  14 tests

12:48:45 PIT >> INFO : MINION : 12:48:45 PIT >> INFO : Dependency analysis reduced number of potential tests by 0

12:48:45 PIT >> INFO : MINION : 12:48:45 PIT >> INFO : 14 tests received

-12:48:45 PIT >> INFO : Calculated coverage in 0 seconds.
12:48:45 PIT >> INFO : Created  2 mutation test units
-12:48:46 PIT >> INFO : Completed in 1 seconds
================================================================================
- Mutators
================================================================================
> org.pitest.mutationtest.engine.gregor.mutators.ConditionalsBoundaryMutator
>> Generated 4 Killed 4 (100%)
> KILLED 4 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.IncrementsMutator
>> Generated 2 Killed 2 (100%)
> KILLED 2 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.ReturnValsMutator
>> Generated 2 Killed 2 (100%)
> KILLED 2 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.MathMutator
>> Generated 4 Killed 4 (100%)
> KILLED 4 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.NegateConditionalsMutator
>> Generated 4 Killed 4 (100%)
> KILLED 4 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
```

Figure 6 The results of the automation mutation via Pitest

4.4 An evaluation of the tested software

The results showed that the testing which has been performed by JOrtho library developers in one of the stages of software development lifecycle, was not very thorough because we were able to found two faults even we have not tested a lot of features. Maybe if they followed Agile Testing approach [11], which main purpose is to fix the errors early in the development process, could significantly increase the quality of the software.

What is more, the functionality which is documented in the JOrtho library documentation is missed, e.g. the word does not have a red highlighting if it starts with a lowercase letter and is a first word of the sentence. However, the acceptability criteria defined earlier in Test Plan chapter were met and the implemented functionality of the JOrtho library could be useful while developing a particular application.

References

[1] "JOrtho - Java Orthography Checker download | SourceForge.net," 2019. [Online]. Available: https://sourceforge.net/projects/jortho/. [Accessed: 25-Mar-2019].

[2] C. Kaner, J. Bach, and B. Pettichord, Lessons Learned in Software Testing: A Context-Driven Approach. Wiley, 2011.

[3] "HunSpell," 2018. .

[4] "Web Spell Checker for Java, IIS ASP .NET, Ajax, JavaScript Forms, Open Source & Royalty Free Spell Check Licenses." [Online]. Available: https://www.jspell.com/. [Accessed: 25-Mar-2019].

[5] P. E. Central, Ed., Introduction to Software Testing. Cambridge University Press Textbooks, 2008.

[6] "Arquillian · Write Real Tests." [Online]. Available: http://arquillian.org/. [Accessed: 01-Apr-2019].

[7] "Mockito framework site." [Online]. Available: https://site.mockito.org/. [Accessed: 01-Apr-2019].

[8] Y. Jia and M. Harman, "IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 1 An Analysis and Survey of the Development of Mutation Testing," 2010.

[9] "PIT Mutation Testing." [Online]. Available: http://pitest.org/. [Accessed: 01-Apr-2019].

[10] "Stryker Mutator." [Online]. Available: https://stryker-mutator.io/. [Accessed: 01-Apr-2019].

[11] "Agile Testing, Agile Testing Methods, Principles and Advantages - ReQtest." [Online]. Available: https://reqtest.com/testing-blog/agile-testing-principles-methods-advantages/. [Accessed: 01-Apr-2019].