Open Assessment

Author:
Y3507677

Module:
Software Testing (SOTE)

April 25, 2019

Contents

1 Introduction

JOrtho is an open source spell-checker that is written using the Java programming language. It is designed around the Swing framework and can be binded to a `JTextPane`, `JEditorPane` or `JTextArea` component. The JOrtho SVN trunk does contain tests, however, a number of them are not passing and they do not have a high test coverage. This report documents the testing of the current revision in the SourceForge SVN repository (r284) and documents how fit for purpose this revision is.

2 Test Plan

2.1 Development Environment

Tests were developed and tested using IntelliJ IDEA 2018.3.5 using Ubuntu 18.04, on a Ryzen 5 1600 @ 3.85 GHz, 8GB DDR4 and a 500GB SSD, as well as being tested on the Computer Science department's computers to ensure that it worked effectively. IntelliJ IDEA Ultimate has been used to implement the tests due to it's seamless integration with JUnit and native code coverage tools. The plugin MetricsReloaded [1] was used for generating statistics about the JOrtho library and aided in deciding what to test.

2.2 Existing Tests

JOrtho already contains a number of tests in its trunk, however, the test `UtilsTest.testRemoveUnicodeQuotation` fails due the `Utils.replaceUnicodeQuotation` not containing a switch statement for the unicode character `U+201F`. Already this is raising alarm bells as this Unicode character has existed since version 1.1.0 (1993) and this SVN revision is from 2016.

In addition to the previous failing test, `MemoryTest.testCreateLanguagesMenu` is very poorly implemented and is unpredictable as to whether or not it will pass or fail. This method aims to detect memory leaks in the application rather poorly; `System.runFinalization()` and `System.gc()` *suggests* that the JVM performs certain tasks, not that it *will*. As such, they cannot be used in this context. In addition to this, `Runtime.freeMemory()` returns an approximation of the memory in use. This test method will pass when the class `AllTests` is run, and fail when run by itself. To get this class to pass by itself the JVM arguments must be changed to `-Xms8m -Xmx128m` - or another suitable starting memory allocation pool size. The testing of these VM arguments did highlight an issue in the testing where if the maximum memory allocation pool size was too small then, of course, a `java.lang.OutOfMemoryError` is thrown but the test still passes.

2.3 Testing Strategies

2.3.1 Expected Behaviour of JOrtho

JOrtho does not include documentation, however, the code itself is not too difficult to understand and some important areas do have comments. In addition to this, there is a sample application that appears to work as expected and demonstrates the binding of the library to a Swing UI textarea. This helps to gain an understanding of how the library works and the expected functionality. JOrtho defines itself as a library that "works with any JTextComponent from the swing framework" [2] and as such, it should be tested on a range of component to ensure that it can be used effectively for the project that management wishes to implement it in. Like traditional spellcheckers, it is expected that once a spelling error has been made it is underlined and possible resolutions are displayed to the user to choose from.

2.3.2 Approaches

A variety of testing strategies exist that could be used to test JOrtho, however, due to the constraints set (8 tests, 10 pages) not all are suitable. The goal of this testing is to check that the library is dependable before using it in multiple projects. A black box testing approach will be taken towards JOrtho due to the lack of documentation. While it is possible that developers could spend a large amount of time attempting to figure out how to correctly use JOrtho, there will be times where the application is run without fully knowing what the outcome will be. Because of this, it is believed that a black box testing strategy is the best approach as it will help to identify both bugs in the software and implementation issues. Helping to identify where the library fails will help to increase the understanding of it.

There are four main ways in which JOrtho could be tested [3]:

- **Critical Functionality**: test cases should be derived that measure individual goals that the software is trying to achieve and then tested using positive (valid) and negative (invalid) data sets, ensuring that the application can handle both data sets.
- **Usability of the Application**: user-based testing. How well a user can interact with the software. The application should be easy to navigate and not confuse the end-user.
- **Performance Testing**: ensuring that the application executes executes in a timely manner; load testing, spike testing, soak testing etc.
- **Look and Feel Testing**: ensuring that the application displays correctly (if applicable) on a variety of machines and that UI components load correctly.

Both usability, and look and feel testing will not be tested as part of this report. JOrtho is a library and the aim of this report is to ensure that it is fit for purpose; the implementor of this library will be required to perform these tests at a later date. It is, however, important to perform some level of performance testing to ensure that the user receives feedback as

soon as possible on their typing. In addition to this, a large focus will be placed on critical functionality testing to ensure that the library does indeed meet what it has set out to achieve.

2.3.3 Existing tests

Appendix 6 table 6 and 7 contains a breakdown of the existing code coverage by package and class respectively. There are no unused classes or methods and the metrics show us that a low percentage (22%) of the total code base is being tested.

Before deciding on which aspects of the library to focus on testing, a true understanding of JOrtho is required and in order to gain this understanding, cyclomatic complexity metrics were generated using the MetricsReloaded plugin. This aids in finding complex classes and methods within the library and where more complex code resides. Figure 1 breaks down the weighted method complexity (WMC) by class for class's that have a WMC of over 10; to view the top results. Table 1 details the top three classes and their weighted method complexity.

Class name	**WMC**	**Lines of code (LOC)**
com.inet.jortho.SpellChecker	656	67
com.inet.jortho.SpellCheckerDialog	243	44
com.inet.jortho.DictionaryBase	213	37

Table 1: Existing package test code coverage in JOrtho

These results have highlighted three core parts to the library. Investigating the top classes by cyclomatic complexity is detailed in figure 2 with the top three methods being detailed in table 2. These three methods are core to the functionality that JOrtho provides and do not currently have any tests which cover them; in fact, none of the `Dictionary` classes have any code coverage apart from the factory. Given how essential they are to the library, it is important that these methods are tested. It is important to test methods with high cyclomatic complexity as they can be a source of confusion for developers, harder to test and more prone to error [4]. McCabe [5] proposed a maximum limit of 10, however, this has proven to be a number that varies in many applications. There are currently seven methods that are on or exceed a value of 10 and even allowing for 1.5 times McCabe's recommendation still results in two methods; the average cyclomatic complexity for the top three offending classes is around 5.

Another metric which could be used to identify which areas of the library to test would be to look at the class change frequency using SVN logs and identify which areas change most frequently. While it would be possible to create these metrics from the repository it is not necessary as it is not a high traffic repository or a particularly complex code-base, it isn't clear whether or not the user `i_net_software` is being used by multiple committers, and there is no additional bug tracking information - that is of any value - that can be included. Mens et al. [6, p. 71] suggest that it is possible to predict areas that contain bugs

or will contain bugs, with the aforementioned information.

Method name	**v(G)**	**Lines of code (LOC)**
DictionaryBase.searchSuggestions (Suggestions, CharSequence, int, int, int)	21	99
Tokenizer.nextInvalidWord()	21	52
AutoSpellChecker.checkElement(Element)	12	43

Table 2: Existing package test code coverage in JOrtho

2.3.4 Testing Goals

Given the expected behaviour of JOrtho, a number of goals can be derived in order to test that the software does actually meet this criteria in an effective fashion. This must be coupled with the goal of ensuring that the library is dependable before committing to a project. The application of JOrtho is something that the user will be inputting and it is paramount that the library is defensive enough to be able to handle poor spelling and rapidly changing text. This means testing *"high-impact problems before low-impact problems"* [7, p. 5] to ensure that the application can handle a range of inputs. Table 1 highlighted that there are three classes at the core of the library and Kaner et al. [7, p. 5] state *"test core functions before contributing functions"* and as such, these classes should investigated for bugs.

Testing will largely focus on maximising bug count, finding defects and investigating the quality of the library. Due to not knowing the target application that this library will be used on, it is not possible to hone in these tests towards a specific purpose and thus the tests will focus on the system as a whole to ensure reliability.

2.3.5 Unit Testing

While every step of testing is important, unit tests will be the core of this testing strategy due to the data-driven application of JOrtho. It is critical to ensure that the library can handle erroneous input. Unit tests allow for testing code in isolation and narrow the scope of the test, however, trivial behaviours such as accessors and mutators should not be tested [8, p. 10]. While there may be tests stored outside of the SVN repository, the lack of tests bundled with the source code leads one to believe that the code was not designed using tests and this can be worrying.

Due to the tight coupling between the components of the library, it does not leave many areas of code that can be unit tested alone and as such, more integration tests are required. Investigations were made in to simulating `DocumentEvents` in order to unit test classes effectively, however, this proved to be an unsuccessful approach. Another option would have been to use reflection to individually test certain core functions, however, this is bad practice. In addition, a number of the methods are not defensively programmed and their

outcomes could be predicted. These tests aim to target sections of the library that have more unpredictable outcomes and will cause frustrations to either the user of the application or to the developers.

2.3.6 Integration Testing

This is a critical part of JOrtho as a number of components are binded together and must work effectively. Interestingly, three of the core classes are highlighted in table 2 and any defects present in this could completely alter the behaviour of the application.

2.3.7 System Testing

System testing is the act of testing the complete library as a whole and can be performed using the provided sample application. A crucial step in the testing steps is to ensure that the UI updates based on the data that has been entered and that all UI components load as expected in the event of a spelling mistake.

2.3.8 Performance Testing

JOrtho is based on Wiktionary and the English library download at the provided revision contains 608,305 entries in total. JOrtho's dictionary is also backed by a Binary Search Tree datastructure that has an average time complexity of `O(log(n))` for access, search, insertion and deletion as well as a worst space complexity of `O(n)`. As a result of this, the application response time could grow in proportion to the length of the entered word. Due to the lack of documentation, there is no information regarding what restrictions are in place for long word lengths. As such, the library should be performance checked to ensure that it can handle a wide range of input lengths.

3 Test Case Specifications

3.1 Test Case 1: Words That Contain Numbers

Test method: `InputTests.testWithNumbers`, (Integration Test)

Spelling checkers, such as those found in Microsoft Word, do not consider a word to be invalid if it starts with a number, but do if it contains numbers in a string that starts with a character. This test seeks to check that JOrtho inhibits a similar behaviour. It is acceptable for an item found in a document to start with a number but not for a word to contain one. This test iterates over a 17 character string and inserts numbers and then asserts against the correct behaviour.

3.2 Test Case 2: Long String Inputs

Test method: `DictionaryTest.testLongAdd`, (Performance Test)

Given that we do not know the application that this library will be used in, it is important to understand how the library will behave when it is presented with a long string. The final application could be a text editor with a spelling checker that checks on demand, or that is disabled for certain file types. If, for instance, a binary file is opened then long strings could be fed to the library and cause the application to become unresponsive for a long period of time. While this is a performance test, it is important to also consider this from a user interaction perspective and ensure that the application does not lock-up for too long. Investigation has revealed that Microsoft Word disables its spell checking functionality for strings over 64 characters in length and it is expected that JOrtho exhibits the same behaviour.

3.3 Test Case 3: Search Suggestions

Test method: `DictionaryBaseTest.testSearchSuggestions`, (Integration Test)

This test will ensure that the spell checker is working correctly when provided with a valid input (non-null) and that suggestions are returned. Following this, it will ensure that the tokenizer is performing its job correctly and not still detecting spelling mistakes. This test will be performed by setting an invalid string in the field and asserting that the dictionary's suggestion feature returns a list of suggested replacements. From here, setting the text area to be one of the suggested words should yield no invalid words within it.

3.4 Test Case 4: Search Suggestion null handling

Test method: `DictionaryBaseTest.testSearchSuggestionsNullInput`, (Unit Test)

It is possible that the library will not be used in the same fashion as the sample applications provided and as such, it is important to test how defensive the library's methods are. In addition to this, it is important to test that the library will be maintainable and not require unnecessary external checks that should have been implemented in the library originally.

While this is a trivial test, it is important that this tests passes as it could result in a `NullPointerException` which would terminate the application if not handled. The method `searchSuggestions` is the second highest method by cyclomatic complexity. If an exception is thrown by a `JTextComponent` after calling the `getText` method then `null` is returned as the value and if not checked then the dictionary will throw an exception.

3.5 Test Case 5: Special Characters

Test method: `InputTests.testSpecialCharacters`, (Integration Test)

Microsoft Word and other popular spell checkers treats any non-alphabetic characters as a break in a word and any two words divided by a special character should be treated as two separate words. It is expected that this functionality will be included in JOrtho so

users are familiar with how the spell checker operates and do not have words missed. This test will be performed by taking two invalid words, a character array of special characters and then joining the two words together with one special character inbetween.

3.6 Test Case 6: Testing Paragraphs and Highlighting

Test method: `HighlighterTest.testParagraph`, (Integration Test)

It is not possible to predict how a user will type in to a text area or how they will use the spell checker. Because of this, it is important to test that the library still works effectively even with sporadic typing. This test aims to check the highlighting functionality and how the library handles paragraphs being split. Ensuring that the user still receives feedback on their writing. If this does not occur, then JOrtho is not achieving what it has set out to do.

3.7 Test Case 7: Web Address Detection

Test method: `TokenizerTest.testWebAddress`, (Integration Test)

This test aims to check that web addresses are detected correctly by the library. A commonly used address (*www.google.com*) will be provided to the tokenizer and it is expected that it is detected as a URL.

3.8 Test Case 8: Ignoring All Occurrences of a String

Test method: manual system test

Given a word that the spell checker believes to be spelt incorrectly, ignore all occurrences of it and check to see if they do stay ignored for the remainder of the session. This feature is particularly useful if the user wishes to ignore a word but not add it to the dictionary; perhaps for just a particular session. Due to the tight coupling of the components that exists within the library, this has to be a manual user test. The expectation here is that following an incorrect spelling being made, the user opens the spelling dialog (F7 key) and clicks "ignore all". Following closing the window, all occurrences of that word should no longer be highlighted red. While there is no documentation stating that this is expected behaviour, it is commonly found behaviour in spelling checkers.

4 Test Results

Test	Pass?	Expected	Actual	Notes
1	Yes	The first string should not be detected by the `Tokenizer`. The remaining strings should be returned as invalid words.	N/A	N/A

Test	Pass?	Expected	Actual	Notes
2	No	Expect the application to gracefully handle long inputs. Or, provide visual feedback to the user if an operation is taking a long period of time to execute.	The add operation does complete, however, it does not provide any feedback to the user and can appear that it has locked up.	If the library was to be used in a code editor, at present, opening a binary file would cause the application to be non-responsive for a very long period of time.
3	Yes	Expect the application to provide a number of suggestions to the test string and for the tokenizer to recognise the spelling mistake. Once a suggestion is chosen and the error is remedied the tokenizer should no longer return any invalid words.	N/A	N/A
4	No	The application should handle the null input and return no suggestions.	`NullPointerException` is thrown instead.	Given the control flow that is found within the application, it is possible for null to be an input to this method and it should have been taken account for.
5	No	Any single input that is separated by a special character should be treated as two separate words. For example, `hello!hello` should be treated as two invalid words, not one.	Fails on: ["], ['], [-], [.], [_], [‘] - Comma-separated values and non-inclusive of the outer brackets.	This behaviour is what is found in most spell checkers.
6	No	It is expected that any paragraph that contains an invalid word, no matter what sequence of events cause a paragraph to be formed, is detected and for the word to be highlighted to the user.	If multiple paragraphs already exist and the user adds a new line between the two paragraphs, if the lower paragraph has an invalid word, the highlight is removed.	N/A
7	No	The URL (www.google.com) should be treated as a web address and not as an invalid word.	The URL is marked as invalid and the only resolution is to add it to the dictionary. This is not normal behaviour.	Well formatted URLs, that include HTTP are treated normally.

Test	Pass?	Expected	Actual	Notes
8	No	Following entering an invalid word, selecting "Ignore all occurrences" should remove the highlight from all occurrences of that word.	Only ignores it for the remainder of the spell checking dialog activity, not in the editor.	

5 Test Summary Report

A number of tests have been conducted to investigate how fit for purpose JOrtho is. While it was not possible to directly unit test a number of the methods, the methods with high cyclomatic complexity that were identified as concerning, were tested. These methods did indeed fail to pass the tests that were conducted. Table 3 details the increase in code coverage with the new tests in place.

Element	Class (%)	Method (%)	Line (%)
com.inet.jortho	61% (34/49)	49% (135/259)	43% (726/1577)
Change with new tests	33% (+15)	25% (+74)	28% (+485)

Table 3: Test coverage with new tests

Table 4 details the branch and line coverage from the tests that were implemented. Due to the limited number of tests available and the high cyclomatic complexity of some of the methods, it was not possible to increase the branch coverage further. A lot of the remaining branches appear to be the result of a poor design structure that is present within the library and this has resulted in such a high cyclomatic complexity. While it would have been possible to increase the test coverage in these classes it was more important to test a wider range of features in JOrtho in an attempt to uncover more serious bugs.

Class	Covered Branches	Missed Branches	Covered Lines	Missed Lines
DictionaryBase	67	11	106	3
Dictionary	20	8	56	37
Suggestions	7	1	16	0
Tokenizer	34	32	64	42
AutoSpellChecker	31	17	74	24

Table 4: Branch and line coverage results for relevant classes

Table 5 details the severity and categories for the results found. While a number of the results are unexpected features of the library, the number of them found from only eight tests leads one to believe that there may be more hidden. In addition to this, such a high level of

unexpected features - which are found in other spell checkers - may lead to an uncomfortable experience using the software; additional user testing will be required in order to confirm this. Testing has revealed three major bugs in the software that could result in potential data loss and invalid results being presented to the user.

Test	Category	Severity
2	Lack of input validation	Major
4	Lack of input validation	Major
5	Unexpected feature or bug	Minor
6	Lack of input validation	Major
7	Unexpected feature or bug	Minor
8	Unexpected feature or bug	Minor

Table 5: Test result classification

Overall it is believed that testing was executed effectively. While some issues were had regarding the desire to unit test certain methods instead of integration test them, the tests were performed effectively. A 75% failure rate in the tests has resulted in valuable feedback regarding the performance of the library and identified areas that suffer from issues. This testing has achieved the goal of whether or not the library is dependable through a range of test strategies. The three classes that were identified has having the highest cyclomatic complexity were used to initiate testing and this has resulted in a significant increase in the code coverage.

While only one of the tests resulted in the application failing, the lack of input validation could lead to an exception at an unpredictable time and result in data loss. Additional documentation would have aided in improving the overall testing strategy as expected behaviour would have been known. Nonetheless, the testing was still executed successfully. It is believed that resolutions to the tests that failed would not require significant changes to the library. One of the most worrying test results was the spell checker running on any length input and the application becoming non responsive. This could very easily be resolved through simply not running the spell checker on a word that is over 64 characters in length.

It is my final recommendation that JOrtho is not used unless either: the developers will update the library and resolve the aforementioned issues, or a fork is made and the library is heavily updated; making it more defensive in how it treats data and for the experience to feel similar to other spell checkers. JOrtho has demonstrated low reliability due to its lack of defensive programming and does not feel like similar spell checking libraries; this could be because of how old the library is and perhaps simply updating it would remove this alien feeling. JOrtho could serve as a potential base for building a more robust system on top of, however, it would still be my recommendation to perhaps use a more advanced library such as Bobbylight's [9] - which still integrates with Swing. In addition to this, I believe that it would be worth investigating other solutions that may be available before committing to using an out-of-date library.

References

[1] B. Leijdekkers, "Metricsreloaded - plugins | jetbrains." [Online]. Available: https://plugins.jetbrains.com/plugin/93-metricsreloaded

[2] "Jortho - java orthography checker." [Online]. Available: https://sourceforge.net/projects/jortho/

[3] S. Dalal and R. S. Chhillar, "Software testing-three p's paradigm and limitations," *International Journal of Computer Applications*, vol. 54, no. 12, p. 49âĂŞ54, 2012. [Online]. Available: https://pdfs.semanticscholar.org/f72a/124e8069515474df410ae4cc883a5fa9c970.pdf

[4] D. R. Wallace, A. H. Watson, and T. J. Mccabe, "Structured testing :," 1996.

[5] T. McCabe, "âĂIJa complexity measure," *IEEE Transactions on Software Engineering*, p. 308âĂŞ320, Dec 1976. [Online]. Available: http://www.literateprogramming.com/mccabe.pdf

[6] T. Mens and S. Demeyer, *Software evolution.* Springer, 2010.

[7] C. Kaner and J. Bach, *Lessons learned in software testing.* Wiley, 2001.

[8] L. Koskela, *Effective unit testing: a guide for Java developers.* Manning, 2013.

[9] Bobbylight, "bobbylight/spellchecker," Dec 2018. [Online]. Available: https://github.com/bobbylight/SpellChecker

6 Appendix

6.1 JOrtho existing test code coverage

Element	Class (%)	Method (%)	Line (%)
com.inet.jortho	44% (22/49)	28% (73/259)	22% (356/1577)

Table 6: Existing package test code coverage in JOrtho

Class	Class (%)	Method (%)	Line (%)
EventTest	100% (1/ 1)	100% (4/ 4)	100% (37/ 37)
RedZigZagPainter	100% (1/ 1)	50% (2/ 4)	18.5% (5/ 27)
MemoryTest	100% (1/ 1)	100% (5/ 5)	93.2% (41/ 44)
CheckerMenu	100% (1/ 1)	100% (2/ 2)	90% (9/ 10)
SpellCheckerOptions	100% (1/ 1)	13.3% (2/ 15)	33.3% (10/ 30)
AllTests	100% (1/ 1)	66.7% (2/ 3)	84.2% (16/ 19)
DefaultMessageHandler	100% (1/ 1)	25% (1/ 4)	30% (3/ 10)
PopupListener	100% (1/ 1)	25% (1/ 4)	30% (3/ 10)
Utils	100% (1/ 1)	15.4% (2/ 13)	15.9% (11/ 69)
LowMemoryArrayList	100% (1/ 1)	80% (4/ 5)	55.6% (10/ 18)
WordIterator	100% (1/ 1)	85.7% (6/ 7)	76.9% (20/ 26)
UtilsTest	100% (1/ 1)	100% (2/ 2)	66.7% (4/ 6)
DictionaryFactory	100% (2/ 2)	63.6% (7/ 11)	46.9% (30/ 64)
SpellChecker	85.7% (6/ 7)	56.1% (32/ 57)	50.6% (133/ 263)
CheckerListener	50% (1/ 2)	10% (1/ 10)	7.8% (7/ 90)
AutoSpellChecker	33.3% (1/ 3)	23.5% (4/ 17)	17% (17/ 100)
AboutDialog	0% (0/ 2)	0% (0/ 3)	0% (0/ 38)
Tokenizer	0% (0/ 1)	0% (0/ 12)	0% (0/ 101)
DictionaryEditDialog	0% (0/ 5)	0% (0/ 16)	0% (0/ 113)
LanguageBundle_fr	0% (0/ 1)	0% (0/ 2)	0% (0/ 12)
FileUserDictionary	0% (0/ 1)	0% (0/ 5)	0% (0/ 30)
LanguageBundle	0% (0/ 1)	0% (0/ 5)	0% (0/ 11)
CheckerPopup	0% (0/ 1)	0% (0/ 1)	0% (0/ 4)
Dictionary	0% (0/ 1)	0% (0/ 14)	0% (0/ 81)
LanguageChangeEvent	0% (0/ 1)	0% (0/ 3)	0% (0/ 5)
Suggestion	0% (0/ 1)	0% (0/ 7)	0% (0/ 11)
AddWordAction	0% (0/ 1)	0% (0/ 3)	0% (0/ 11)
SpellCheckerDialog	0% (0/ 5)	0% (0/ 21)	0% (0/ 159)
DictionaryBase	0% (0/ 1)	0% (0/ 10)	0% (0/ 104)
Suggestions	0% (0/ 1)	0% (0/ 4)	0% (0/ 13)

Table 7: Existing class code coverage in JOrtho

6.2 JOrtho statistics

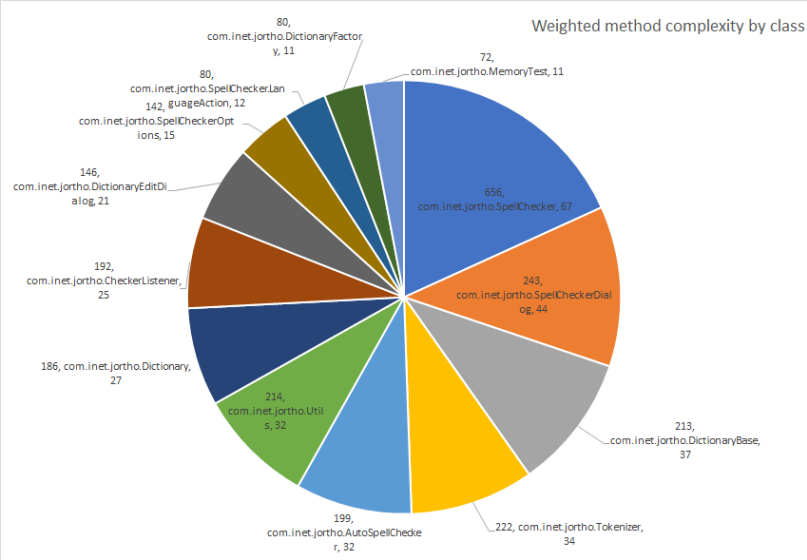

Figure 1: Weighted mean complexity by class for JOrtho. Labels are in the format of (number of lines), (class name), (complexity)

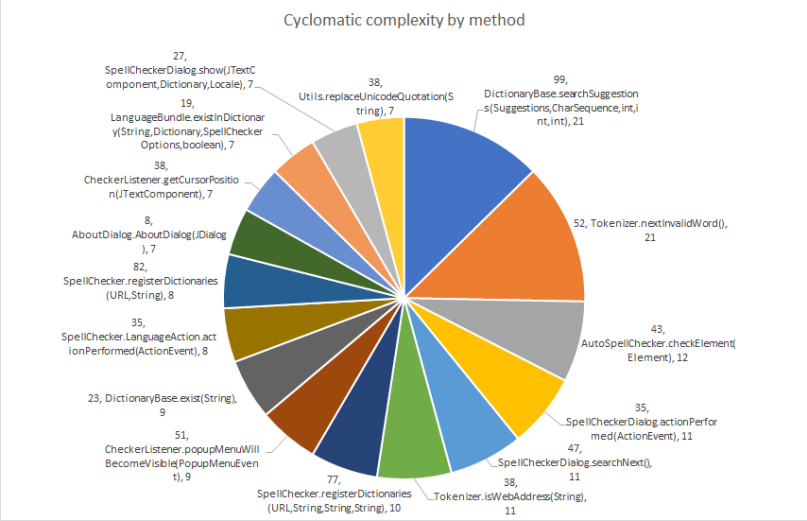

Figure 2: Cyclomatic complexity by method for JOrtho. Labels are in the format of (total cyclomatic complexity), (method name), (complexity)

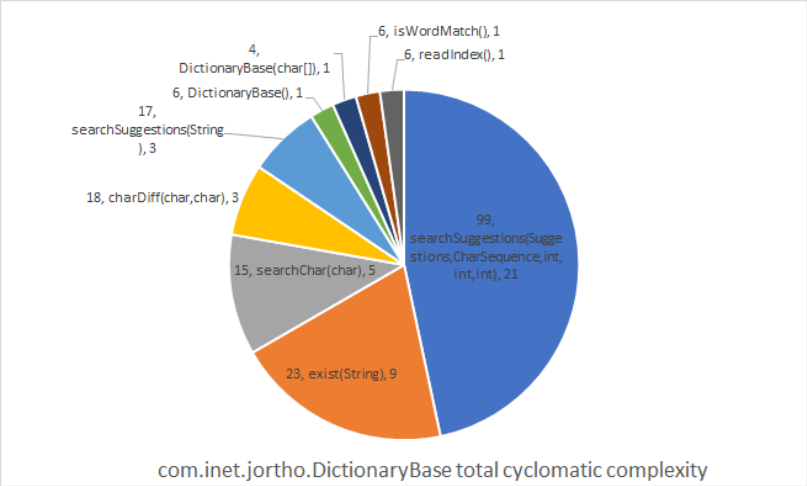

Figure 3: Cyclomatic complexity by method for the DictionaryBase class. Labels are in the format of (total cyclomatic complexity), (method name), (complexity)

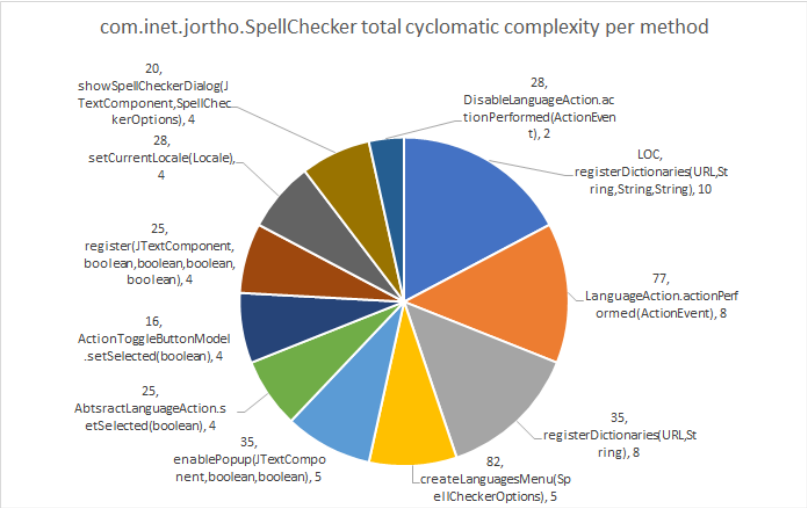

Figure 4: Cyclomatic complexity by method for the SpellChecker class. Labels are in the format of (total cyclomatic complexity), (method name), (complexity)

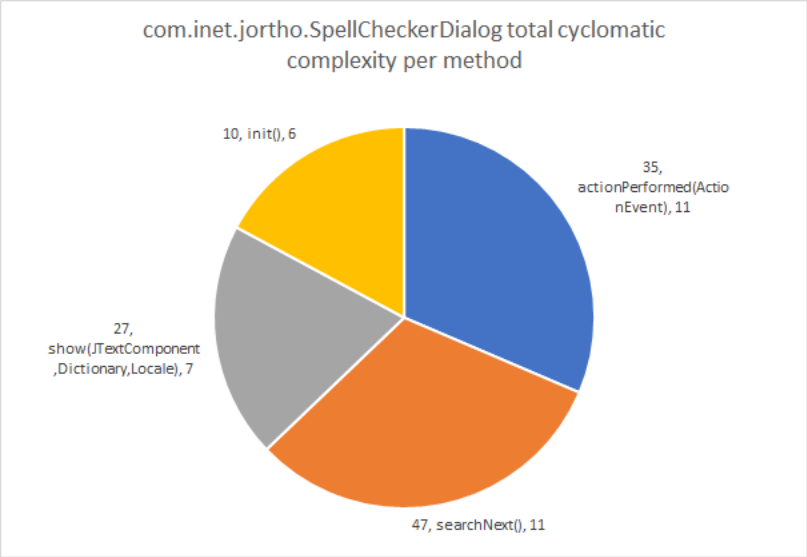

Figure 5: Cyclomatic complexity by method for the SpellCheckerDialog class. Labels are in the format of (total cyclomatic complexity), (method name), (complexity)

End of Examination