# Metrics report

## Introduction

In this report there will be an analysis of the complexity metrics regarding the JabRef project. There will be an overall exposure of the said metrics on various levels (project, modules, packages, classes & methods), but the focus will be on the **classes** and **methods** levels (mainly the latter) where a discussion of the results will take place and a correlation of those results with code smells present in the code will be made.

Across the report various abbreviations will appear:

- v(G)avg – average cyclomatic complexity

- v(G)tot – total cyclomatic complexity

- OCavg – average operation complexity

- OCmax – maximum operation complexity

- WMC – weighted method complexity

- CogC – cognitive complexity

- ev(G) – essential cyclomatic complexity

- iv(G) – design complexity

- v(G) – cyclomatic complexity

All the metrics data was extracted using the IntelliJ IDEs plugin MetricsReloaded.

## Complexity

Complexity is normally measured with cyclomatic complexity, which was developed to determine the stability and level of confidence in a program. In more detail, it measures the number of linearly independent paths through a program module.  
A lower cyclomatic complexity in a program means it is easier to test.

## Project



The project as whole doesn’t exceed the metric thresholds defined by the plugin, presenting an average cyclomatic complexity of 1,79.

### Modules



The modules also don’t exceed the metric thresholds, presenting the same average cyclomatic complexity as the project. Even so, we can start to identify some modules that exceed that average, as can be seen with the boxplot on the right.

### Packages



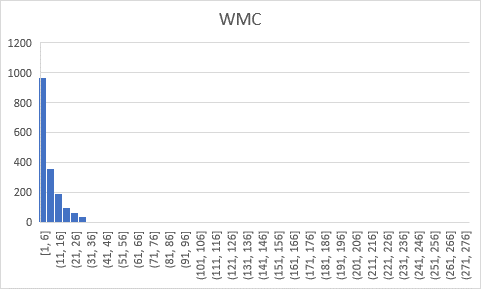
The packages, like the project and the modules, don’t exceed the thresholds defined by the plugin, maintaining an average cyclomatic complexity of 1,79. But when comparing their boxplot to the modules’, there appears to be some modules with a significantly higher average cyclomatic complexity than the average, reaching a maximum of 4,49 among them.

### Classes



Note: when dealing with the classes, the plugin defines two clear thresholds for OCavg and WMC: 3 and 30, respectively.

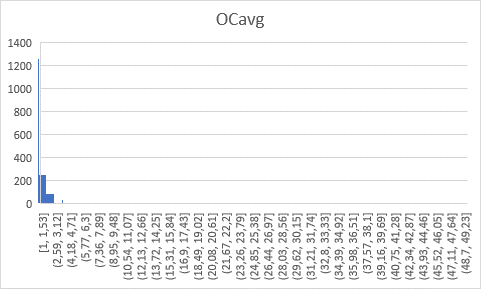
Let us look at the distribution of the weight of all the different classes:



We can see that, when compared to the threshold, most of the classes are within the boundary stipulated by the plugin, but there are still some with extreme values (outliers). Those outliers represent classes that have a high sum of complexities of their methods, so overall the class itself has a high complexity and may be difficult to get into when compared to those with a lower WMC value.

That doesn’t mean that evaluating the WMC of a class is the best way to approach its complexity since that same complexity can be very well distributed by a lot of different operations that make the code itself more digestible and easier to manipulate if needed. Of course, this last statement doesn’t apply to the classes where one method by itself has a score higher than 30.

Therefore, let us analyze the distribution of the average operation complexity of all the different classes:



It is evident that most of the average operation complexity values are below the average, but the outliers are still present here with values that go up to 49,0. The classes with a high average operation complexity are the problematic ones: there is most likely a bad distribution of the complexity into different methods, increasing the chance of a code smell being present then.

The three classes where the OCavg is the highest are:

- org.jabref.gui.fieldeditors.FieldNameLabel (49,00)

- org.jabref.logic.layout.format.RTFChars (25,00)

- org.jabref.logic.layout.format.GetOpenOfficeType (15,00)

### Methods

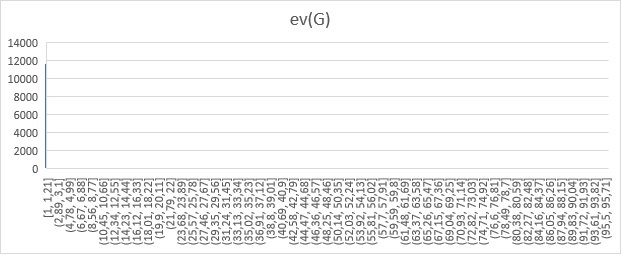
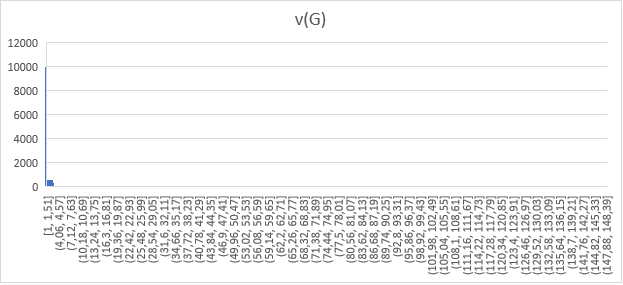


Note: when dealing with the methods, the plugin defines clear thresholds for all its subjects of evaluation: 15, 3, 8, 30, in order. The focus will be in the ev(G) (3) and v(G) (30).

ev(G) is the essential cyclomatic complexity that indicates how much complexity is left after removing well-structured complexity (for example a for loop where a condition is stated at the start of the loop). It mostly tells us how easy a method is to understand.

v(G) is the normal cyclomatic complexity stated at the beginning of the report.

Knowing this, we can say that if a method as a high v(G) but low ev(G), then it is difficult to test as a whole, but can be broken up into smaller operations to make the testing easier and, therefore, easier to refactor in case of the presence of a code smell.

Kicking off by examining the essential and non essential cyclomatic complexities ditributions of all of the methods:

We can see that in both cases, most of the methods have complexities below their respective thresholds and in range of their average. But, as expected, we can identify the existence of outliers on both histograms, leading to belive that some methods have very high complexity (way above the thresholds already mentioned), with 96 for ev(G) and 148 for v(G).

We can identify the top three methods with the highest complexities of each histogram, making sure to present both values for the methods of both histograms:



In both histograms, there are two methods present in both above tables, making it evident that they are the methods more difficult to test and understand. Both also belong in the classes where OCavg was the highest.

Uma imagem com texto

Descrição gerada automaticamente

Uma imagem com texto

Descrição gerada automaticamente

By looking at those methods closely we can see that the complexity is due to the existence of multiple ifs, a switch statement or a combination of both. Although the first method is understandabily complex due to the high number of special characters, there is probably no other way to simplify it due to its nature, but the second is probably an example of a Switch Statement code smell due to the existence of three different if statements, each with a different switch statement.

Switch Statement code smells should be a common ocurrence when there is a high complexity method since the easiest way to increase the complexity of a method is by adding ifs and/or switch statements. The correlation above is a good example of that.

## Conclusion

In conclusion, complexity metrics are a very useful way to evaluate code and to identify problem spots where the occurrence of code smells might be happening. When it comes to JabRef and its complexity, most of the project has a complexity within the thresholds specified by the plugin, but there are still some outlier classes and methods that are outside those values, being good candidates for trouble spots where code smells are probably happening.