Lines of code Metrics



Pedro Reis 58751

**Introduction**

Lines of Code metrics shows how many lines of code there is in your project, interface, module, package, class, method or by file type. This metric simply counts all lines of code or only commented lines of code, Javadoc lines of code, non-comment lines of code, etc. However, this metrics do not take into account the intelligence content and the layout of content and because of that, the LOC metrics are better to monitor the size of the code units and should be avoided when it comes to software estimation because there are better options for it. For example, productivity cannot be measured by LOC because 100 lines of code of one complex algorithm may take the same time of work to do as a system of 10000 lines. Also, this metrics isn’t linear because when using software industry productivity averages a 10000 LOC system would require 13.5 staff months. If effort increased linearly, a 100000 LOC system would require 135 staff months. But it actually requires 170 staff months and the error size of 35 staff months is really significant because you had a budget for only 135 staff months, and you will still need more 35 staff months to finish the project.

**Parameters**

1. Method – When evaluating each method from a program, the structure of the assessment is: Location of method + CLOC + JLOC + LOC + NCLOC + RLOC

Where:

* CLOC – Counts the number of Commented lines of code in the method.
* JLOC – Counts the Javadoc lines of code in the method.
* LOC – Counts the Lines of code in the method.
* NCLOC – Counts the Non-comment lines of code in the method.
* RLOC – Calculates the percentage of lines of code of the method relative to the lines of code of the class where it belongs to.

1. Class – When evaluating each class from a program, the structure of the assessment is: Location of class + CLOC + JLOC + LOC

Where:

* CLOC – Counts the number of Commented lines of code in the class.
* JLOC – Counts the Javadoc lines of code in the class.
* LOC – Counts the Lines of code in the class.

1. Interface – When evaluating each interface from a program, the structure of the assessment is: Location of interface + CLOC + JLOC + LOC + NCLOC

Where:

* CLOC – Counts the number of Commented lines of code in the interface.
* JLOC – Counts the Javadoc lines of code in the interface.
* LOC – Counts the Lines of code in the interface.
* NCLOC – Counts the Non-comment lines of code in the interface.

1. Package – When evaluating each package from a program, the structure of the assessment is: Location of package + CLOC + CLOC (rec) + JLOC + JLOC (rec) + LOC, LOC (rec) + LOCp + LOCp (rec) + LOCt + LOCt (rec) + NCLOC + NCLOCp + NCLOCp (rec) + NCLOCt

Where:

* CLOC – Counts the number of Commented lines of code in the package.
* CLOC (rec) - Counts the number of Commented lines of code recursively in the package.
* JLOC – Counts the Javadoc lines of code in the package.
* JLOC (rec) - Counts the number of Javadoc lines of code recursively in the package.
* LOC – Counts the Lines of code in the package.
* LOC (rec) - Counts the number of lines of code recursively in the package.
* LOCp – Counts the number of lines of product code in the package.
* LOCp (rec) - Counts the number of lines of product code recursively in the package.
* LOCt – Counts the number of lines of test code in the package.
* LOCt (rec) – Counts the number of lines of test code recursively in the package.
* NCLOC – Counts the Non-comment lines of code in the package.
* NCLOCp – Counts the Non-comment lines of product code in the package.
* NCLOCp (rec) – Counts the Non-comment lines of product code recursively in the package.
* NCLOCt – Counts the Non-comment lines of test code in the package.

1. Module – When evaluating each module from a program, the structure of the assessment is: Location of module + JLOC + L (CSS) + L (Groovy) + L (HTML) + L(J) + L(JS) + L(JSP) + L(KT) + L(XML) + LOC + LOCp + LOCt + NCLOC + NCLOCp + NCLOCt

Where:

* JLOC – Counts the Javadoc lines of code in the module.
* L(CSS) – Counts the Lines of CSS in the module.
* L(Groovy) - Counts the number of lines of Groovy in the module.
* L(HTML) – Counts the number of lines of HTML in the module.
* L(J) – Counts the number of lines of Java in the module.
* L(JS) – Counts the number of lines of JavaScript in the module.
* L(JSP) – Counts the number of lines of JSP in the module.
* L(KT) – Counts the number of lines of Kotlin in the module.
* L(XML) – Counts the number of lines of XML in the module.
* LOC – Counts the number of lines of code in the module.
* LOCp – Counts the number of lines of product code in the module.
* LOCt – Counts the number of lines of test code in the module.
* NCLOC – Counts the Non-comment lines of code in the module.
* NCLOCp – Counts the Non-comment lines of product code in the package.
* NCLOCt – Counts the Non-comment lines of test code in the package.

1. File type – When evaluating each file type of a program, the structure of the assessment is: Location of package + LOC + NCLOC

Where:

* LOC – Counts the Lines of code of the file type.
* NCLOC – Counts the Non-comment lines of code of the file type.

1. Project – When evaluating the project of a program, the structure of the assessment is: project + CLOC + JLOC + L (CSS) + L (Groovy) + L (HTML) + L(J) + L(JS) + L(JSP) + L(KT) + L(XML) + LOC + LOCp + LOCt + NCLOC + NCLOCp + NCLOCt

Where:

* CLOC – Counts the number of Commented lines of code in the project.
* JLOC – Counts the Javadoc lines of code in the project.
* L(CSS) – Counts the Lines of CSS in the project.
* L(Groovy) - Counts the number of lines of Groovy in the project.
* L(HTML) – Counts the number of lines of HTML in the project.
* L(J) – Counts the number of lines of Java in the project.
* L(JS) – Counts the number of lines of JavaScript in the project.
* L(JSP) – Counts the number of lines of JSP in the project.
* L(KT) – Counts the number of lines of Kotlin in the project.
* L(XML) – Counts the number of lines of XML in the project.
* LOC – Counts the number of lines of code in the project.
* LOCp – Counts the number of lines of product code in the project.
* LOCt – Counts the number of lines of test code in the project.
* NCLOC – Counts the Non-comment lines of code in the project.
* NCLOCp – Counts the Non-comment lines of product code in the project.
* NCLOCt – Counts the Non-comment lines of test code in the project.

**Analysis**

Method

The LOC metrics presents an average of lines of code for the methods of 10,43 which means the method size of the program perfectly fits between the recommended range that should be 4 to 40 lines. The commented lines average is 1,3 which means 12,5% of the methods are commented lines and this ratio could be improved to 30% for example, in order to have better explained methods.

Despite having a good average, there are some trouble spots that we can identify and there’s one particular method which stands out from the rest for having 429 lines of code which is org.jabref.logic.bst.VM.VM(CommonTree) and it has been identified as a long method code smell.

Class

The LOC metrics presents an average of lines of code for the classes of 75,84 which means the class size of the program perfectly fits between the recommended range that should be 4 to 400 lines. The commented lines average is 13,21 which means 17,5% of the classes are commented lines and this ratio could be improved to fit inside the perfect range which is 30 to 75% of commented lines per class because such a lower percentage of comments per class could mean the class is very trivial or poorly explained. The relative lines of code average is 5% which means on average each method corresponds to 5% of the lines of code of a class. This ratio seems good but there is not any information online to confirm this.

Despite having a good average, there are some trouble spots that we can identify and there’s one particular class which stands out from the rest for having 2291 lines of code which is org.jabref.preferences.JabRefPreferences and it has been identified as a large class code smell. There are other large classes for example the org.jabref.logic.citationkeypattern.BracketedPattern that we have also identified as a large class code smell and org.jabref.logic.util.strings.HTMLUnicodeConversionMaps that with 824 commented code lines it is more like a document. Regarding the RLOC metric, it is possible to identify some methods that occupy over 90% of a class, for example org.jabref.model.strings.UnicodeToReadableCharMap.UnicodeToReadableCharMap() which has a 99% RLOC but this one is not a code smell because it is a map constructor. There are others methods like org.jabref.logic.layout.format.RemoveLatexCommandsFormatter.format(String) that problematic because they are a long method code smell and they should be split.

Interface

The LOC metrics presents an average of lines of code for the interfaces of 22.03, I didn’t find any information regarding the ideal range of an interface, but the size of the interfaces seems good. The commented lines average is 12,11 which means 55% of the interface are commented lines and this ratio seems to be perfect for an interface since it means everything is well explained.

For the trouble spots, there is one interface which may have too many comments since it is composed by 83% of commented lines of code but since this is an interface it may not be worrisome but could mean that this interface is way too complex. There is also an interface

Package

The LOC metrics presents an average of lines of code for the packages of 821,74 and even though the metrics reloaded does not show the averages for the commented code lines counted recursively for each package, I did the average manually and ended up with an average of 166,3 and this means that we have around 20% of commented lines in all the packages.

Module

Not much to analyse here, only that the biggest modules are JabRef and JabRef.buildSrc.main and that the JabRef.buildSrc.main is where most of java code lines are.

FileType

This file type LOC metrics is a good way to see how many lines of code of each file type is the program composed by. Most of the program is made by Files supported via TextMate bundles lines of code, with a significant amount of JAVA lines of code too and some other minor filetypes like XML and others.

Project

The LOC metrics of the whole project allow us to see the dimension of this project, and with 1113118 lines of code and 171743 lines of product code I can conclude this is a big size project. Most of the lines of product code are made in JAVA and we also have a significant amount of XML lines. There is 21373 comment lines and this corresponds to 12,44% of commented lines for all the lines of product code.

**Conclusion**

In conclusion, the most important lines of code metrics to look at are the method, class and interface, and thanks to this metrics we have been able to identify some classes and methods that were too big and needed to be split but also classes and methods that lacked comments or had simply too many comments. Despite this, the overall project is great and fits perfectly inside the recommended range for the classes and methods. LOC metrics are a simple metric and should be used primarily to monitor the size of the code units and to identify the code smells that are related to the size of the code units.