**José Pereira 55204**

**Lines of Code Metrics**

**Method metrics:**

A screenshot of a computer

Description automatically generated

**Explanation of Method Metrics:**

* **CLOC (Comment Lines of Code):** Indicates the number of lines that are comments within method bodies. This metric helps assessing the level of documentation and comments in the code.
* **JLOC (Java Lines of Code):** Refers to the number of Java lines of code, excluding comments and blank lines, within method bodies.
* **LOC (Lines of Code):** Total number of lines of code, including comments and blank lines within method bodies.
* **NCLOC (Non-Comment Lines of Code):** The number of lines of actual code without comments within method bodies.
* **RLOC (Relative Lines of Code):** Percentage of the lines of code relative to the total lines of code within method bodies. It’s shown as 1.63%, which might indicate the relative size of method bodies to the entire codebase.

**Example of a method with high LOC and NCLOC**



* This means that perhaps the method is too complex, doing too much (God Method code smell) and could benefit from refactoring.

**Example of a method with high RLOC**



* This could also mean that refactoring is necessary since 88.10% of the lines of code in the method’s class are within that method.

**Class metrics:**

A screenshot of a computer

Description automatically generated

* **CLOC (Comment Lines of Code):** Reflects the number of lines of comments within classes, indicating how well-documented the classes are.
* **JLOC (Java Lines of Code):** Total number of Java statement lines in classes, not including comments and blank lines.
* **LOC (Lines of Code):** Total number of lines that make up the classes, including code, comments, and blank lines.
* Similar to method metrics, high LOC in a class may suggest that a class is doing too much, potentially violating the Single Responsibility Principle.
* An average CLOC that is significantly lower than the average LOC might suggest under-documentation, whereas a very high average might imply over-commenting which could also be a maintenance issue if comments are not kept up-to-date with changes in code.
* High JLOC can indicate complex classes that may be difficult to maintain or extend.
* Classes with high LOC could be symptomatic of 'Large Class' or 'God Class' code smells, where a class has too many responsibilities.
* Excessive commenting might be indicative of 'Comment Smells,' where the code is not self-explanatory and relies too much on comments for clarity.

These class-level metrics help in assessing the complexity and maintainability of classes.

They can provide insights into the need for refactoring at the class level, such as splitting up classes that are too large or enhancing those with inadequate documentation.

Tracking these metrics over time can help in identifying trends, such as a gradual increase in class size, which could lead to maintainability issues down the line.

**Interface metrics:**

A screenshot of a computer

Description automatically generated

**CLOC (Comment Lines of Code):** The number of lines of comments within interfaces, giving an insight into the documentation level of the interfaces.

**JLOC (Java Lines of Code):** Represents the number of effective Java statement lines in interfaces, not including comments or blank lines.

**LOC (Lines of Code):** The total line count for interfaces, encompassing code, comments, and blank lines.

**NCLOC (Non-Comment Lines of Code):** The count of actual code lines within interfaces, excluding comments.

* Interfaces with an unusually high LOC may be attempting to define too much functionality, potentially violating interface segregation principles.
* A high CLOC relative to LOC could indicate that the interfaces are well-documented, which is usually positive unless the comments are excessively verbose or not meaningful.
* An average NCLOC of 7.37 suggests that the interfaces are quite slim in terms of actual code, which is generally a good practice in interface design.
* Large interfaces (indicated by high LOC) might be a sign of 'Interface Bloat', where an interface contains more methods than its implementers actually require.
* If the CLOC is very low, it could suggest that the interfaces lack adequate documentation, which can make them difficult to understand and implement correctly.
* Interface metrics can be particularly useful in identifying the need for interface segregation or splitting up complex interfaces.
* The balance between CLOC and NCLOC can guide developers on how well the interfaces are explained, which is critical since interfaces define core contracts in the codebase.

These metrics, when tracked over time, can also indicate whether the complexity of interfaces is increasing, which might lead to a more rigid and less maintainable system.

**Package metrics:**

A screenshot of a computer

Description automatically generated

**CLOC (Comment Lines of Code):** It shows zero comment lines of code at the package level, which could be due to the packaging structure not containing comments or it not being the typical place where comments are expected or extracted from.

**LOC (Lines of Code):** The total lines of code in packages, which include everything (comments, whitespace, and actual code).

**NCLOC (Non-Comment Lines of Code):** Represents the number of executable or logical lines of code, excluding comments and empty lines.

* The absence of CLOC at the package level isn't necessarily an issue, but it could suggest a lack of package-level documentation. This might be a point to consider for improvement.
* The average NCLOC at the package level suggest sizable packages. It might indicate that the packages are doing too much or are too large, which could be a sign of poor modularity or low cohesion.
* If certain packages have a particularly high NCLOC compared to others, it might indicate a "Large Class" smell at the package level, meaning that the package could be trying to do too much and might benefit from being broken.

These metrics provide insight into the size and complexity of packages. Large packages can be more difficult to maintain and understand. They can also indicate the scope of changes when packages undergo revisions, affecting testing and integration efforts.

Tracking the evolution of these metrics over time can help in assessing whether the project's modularity and structure are improving or deteriorating, which can guide refactoring efforts.

**Module metrics:**

A screenshot of a computer

Description automatically generated

**JLOC:** This stands for Java Lines of Code. Here, we see 78,867 which likely indicates the number of lines of Java code across the module.

**L(Groovy):** This would represent lines of Groovy code, which is another JVM language. It shows 0, indicating there's no Groovy code in the module.

**L(HTML):** Lines of HTML code, which are 19,605. This indicates that there's a substantial amount of HTML, possibly suggesting that this module contains web interface elements.

**L(J):** This stands for lines of Java code including comments and blank lines.

**L(KT):** Represents lines of Kotlin code, which there aren’t.

**L(XML):** Represents lines of XML code.

**LOC:** This is the total Lines of Code and is very high at 532,580, suggesting a large module with a lot of code.

**NCLOC**: Non-Comment Lines of Code are at 350,658, indicating the actual lines of code minus comments, which is still quite substantial.

* With such a high count of lines of code, this module may be quite complex, potentially housing a diverse range of functionalities. This could make maintenance and understanding more difficult.
* If the module has this much code, it's worth examining whether it adheres to the Single Responsibility Principle. Modules should ideally have one clear purpose.
* With 19,721 lines of test code, we can infer that there is a substantial amount of testing, but we would need to compare the amount of test code to the amount of production code to determine if test coverage is adequate.
* The difference between LOC and NCLOC could give an indication of how much the module is documented. Proper documentation is crucial for maintainability, especially in large codebases.
* Potential for Code Smells: Large modules can sometimes be indicative of "God Objects" or classes that have too many responsibilities. They can become difficult to manage and are prone to errors.

**File type metrics:**

A screenshot of a computer

Description automatically generated

**LOC (Lines of Code):** This indicates the total number of lines of code in the project, including comments, whitespace, and actual code lines. The total LOC here is 532,580.

**NCLOC (Non-Comment Lines of Code):** This represents the number of lines that are not comments or blank lines, which typically equates to the amount of "actual" code. The total NCLOC is 350,658.

**Project metrics:**

A screenshot of a computer

Description automatically generated

**CLOC (Comment Lines of Code):** This metric, showing 98,103 lines, suggests how much of the codebase is documented through comments.

**JLOC:** Stands for Java Lines of Code. Here, we see 78,867 which indicates the number of lines of Java code across the project.

**L (Groovy):** Since this is at 0, it indicates that there are no lines of Groovy code.

**LOC (Lines of Code):** The total lines, 532,580, give you a rough measure of the project's size.

**LOCp (Productive Lines of Code):** The 212,191 lines here are directly contributing to the functionality of the software product.

**LOCt (Test Lines of Code):** With 19,721 lines of test code, it shows a commitment to testing in the project. A high ratio of test code to productive code can be a good sign, indicating a dedication to software quality and stability.

**NCLOC (Non-Comment Lines of Code):** 350,658 lines of executable code without comments. This figure helps in understanding the volume of the code that actually performs the operations and functionalities of the project.

**NCLOCp (Non-Comment Lines of Code for the Product):** The 128,110 lines here shows how much of the codebase is non-comment and non-test code. This helps identify the core of the project's functionality.

**NCLOCt (Non-Comment Lines of Code for Tests):** Finally, the 16,442 lines indicate how much test code there is excluding comments. It can be used to assess the quality and depth of testing in relation to the productive code.