Metric Definitions

Complexity

Complexity (`complexity`)
It is the Cyclomatic Complexity calculated based on the number of paths through the code. Whenever the control flow of a function splits, the complexity counter gets incremented by one. Each function has a minimum complexity of 1. This calculation varies slightly by language because keywords and functionalities do.

Language-specific details

Language	Notes
Java	Keywords incrementing the complexity: `if`, `for`, `while`, `case`, `catch`, `throw`, `&&`, `||`, `?`

Cognitive Complexity (`cognitive_complexity`)
How hard it is to understand the code's control flow. See the Cognitive Complexity White Paper for a complete description of the mathematical model applied to compute this measure.

Duplications

Duplicated blocks (`duplicated_blocks`)
Number of duplicated blocks of lines.

Language-specific details

For a block of code to be considered as duplicated:

Java projects:
There should be at least 10 successive and duplicated statements whatever the number of tokens and lines. Differences in indentation and in string literals are ignored while detecting duplications.

Duplicated files (`duplicated_files`)
Number of files involved in duplications.

Duplicated lines (`duplicated_lines`)
Number of lines involved in duplications.

Duplicated lines (%) (`duplicated_lines_density`)
= `duplicated_lines` / `lines` * 100

Issues

New issues (`new_violations`)
Number of issues raised for the first time in the New Code period.

New xxx issues (`new_xxx_violations`)
Number of issues of the specified severity raised for the first time in the New Code period, where xxx is one of: `blocker`, `critical`, `major`, `minor`, `info`.

Issues (`violations`)
Total count of issues in all states.

xxx issues (`xxx_issues`)
Total count of issues of the specified severity, where xxx is one of: `blocker`, `critical`, `major`, `minor`, `info`.

False positive issues (`false_positive_issues`)
Total count of issues marked False Positive

Open issues (`open_issues`)
Total count of issues in the Open state.

Confirmed issues (`confirmed_issues`)
Total count of issues in the Confirmed state.

Reopened issues (`reopened_issues`)
Total count of issues in the Reopened state

Maintainability

Code Smells (`code_smells`)
Total count of Code Smell issues.

New Code Smells (`new_code_smells`)
Total count of Code Smell issues raised for the first time in the New Code period.

Maintainability Rating (`sqale_rating`)
(Formerly the SQALE rating.) Rating given to your project related to the value of your Technical Debt Ratio. The default Maintainability Rating grid is:

A=0-0.05, B=0.06-0.1, C=0.11-0.20, D=0.21-0.5, E=0.51-1

The Maintainability Rating scale can be alternately stated by saying that if the outstanding remediation cost is:

- <=5% of the time that has already gone into the application, the rating is A
- between 6 to 10% the rating is a B
- between 11 to 20% the rating is a C
- between 21 to 50% the rating is a D
- anything over 50% is an E

Technical Debt (`sqale_index`)
Effort to fix all Code Smells. The measure is stored in minutes in the database. An 8-hour day is assumed when values are shown in days.

Technical Debt on New Code (`new_technical_debt`)
Effort to fix all Code Smells raised for the first time in the New Code period.

Technical Debt Ratio (`sqale_debt_ratio`)
Ratio between the cost to develop the software and the cost to fix it. The Technical Debt Ratio formula is:
`Remediation cost / Development cost`
Which can be restated as:
`Remediation cost / (Cost to develop 1 line of code * Number of lines of code)`
The value of the cost to develop a line of code is 0.06 days.

Technical Debt Ratio on New Code (`new_sqale_debt_ratio`)
Ratio between the cost to develop the code changed in the New Code period and the cost of the issues linked to it.

Quality Gates

Quality Gate Status (`alert_status`)
State of the Quality Gate associated to your Project. Possible values are : `ERROR`, `OK` WARN value has been removed since 7.6.

Quality Gate Details (`quality_gate_details`)
For all the conditions of your Quality Gate, you know which condition is failing and which is not.

Reliability

Bugs (`bugs`)
Number of bug issues.

New Bugs (`new_bugs`)
Number of new bug issues.

Reliability Rating (`reliability_rating`)
A = 0 Bugs
B = at least 1 Minor Bug
C = at least 1 Major Bug
D = at least 1 Critical Bug
E = at least 1 Blocker Bug

Reliability remediation effort (`reliability_remediation_effort`)
Effort to fix all bug issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days.

Reliability remediation effort on new code(`new_reliability_remediation_effort`)
Same as *Reliability remediation effort* but on the code changed in the New Code period.

Security

Vulnerabilities (`vulnerabilities`)
Number of vulnerability issues.

New Vulnerabilities (`new_vulnerabilities`)
Number of new vulnerability issues.

Security Rating (`security_rating`)
A = 0 Vulnerabilities
B = at least 1 Minor Vulnerability
C = at least 1 Major Vulnerability
D = at least 1 Critical Vulnerability
E = at least 1 Blocker Vulnerability

Security remediation effort (`security_remediation_effort`)
Effort to fix all vulnerability issues. The measure is stored in minutes in the DB. An 8-hour day is assumed when values are shown in days.

Security remedation effort on new code(`new_security_remediation_effort`)
Same as *Security remediation effort* but on the code changed in the New Code period.

Size

Classes (`classes`)
Number of classes (including nested classes, interfaces, enums and annotations).

Comment lines (`comment_lines`)
Number of lines containing either comment or commented-out code.

Non-significant comment lines (empty comment lines, comment lines containing only special characters, etc.) do not increase the number of comment lines.

The following piece of code contains 9 comment lines:

```
/**                                  +0 => empty comment line
 *                                   +0 => empty comment line
 * This is my documentation          +1 => significant comment
 * although I don't                  +1 => significant comment
 * have much                         +1 => significant comment
 * to say                            +1 => significant comment
 *                                   +0 => empty comment line
 ***************************         +0 => non-significant comment
 *                                   +0 => empty comment line
 * blabla...                         +1 => significant comment
 */                                  +0 => empty comment line

/**                                  +0 => empty comment line
 * public String foo() {             +1 => commented-out code
 *   System.out.println(message);    +1 => commented-out code
 *   return message;                 +1 => commented-out code
 * }                                 +1 => commented-out code
 */                                  +0 => empty comment line
```

Language-specific details

Language	Note
Java	File headers are not counted as comment lines (becuase they usually define the license).

Comments (%) (`comment_lines_density`)
Density of comment lines = Comment lines / (Lines of code + Comment lines) * 100

With such a formula:

- 50% means that the number of lines of code equals the number of comment lines
- 100% means that the file only contains comment lines

Directories (`directories`)
Number of directories.

Files (`files`)
Number of files.

Lines (`lines`)
Number of physical lines (number of carriage returns).

Lines of code (`ncloc`)
Number of physical lines that contain at least one character which is neither a whitespace nor a tabulation nor part of a comment.

Language-specific details

Language	Note
COBOL	Generated lines of code and pre-processing instructions (`SKIP1`, `SKIP2`, `SKIP3`, `COPY`, `EJECT`, `REPLACE`) are not counted as lines of code.

Lines of code per language (`ncloc_language_distribution`)
Non Commenting Lines of Code Distributed By Language

Functions (`functions`)
Number of functions. Depending on the language, a function is either a function or a method or a paragraph.

Language-specific details

Language	Note
Java	Methods in anonymous classes are ignored.

Projects (`projects`)
Number of projects in a Portfolio.

Statements (`statements`)
Number of statements.

Tests

Condition coverage (`branch_coverage`)
On each line of code containing some boolean expressions, the condition coverage simply answers the following question: 'Has each boolean expression been evaluated both to true and false?'. This is the density of possible conditions in flow control structures that have been followed during unit tests execution.

```
Condition coverage = (CT + CF) / (2*B)
```

where

- CT = conditions that have been evaluated to 'true' at least once
- CF = conditions that have been evaluated to 'false' at least once
- B = total number of conditions

Condition coverage on new code (`new_branch_coverage`)
Identical to Condition coverage but restricted to new / updated source code.

Condition coverage hits (`branch_coverage_hits_data`)
List of covered conditions.

Conditions by line (`conditions_by_line`)
Number of conditions by line.

Covered conditions by line (`covered_conditions_by_line`)
Number of covered conditions by line.

Coverage (`coverage`)
It is a mix of Line coverage and Condition coverage. Its goal is to provide an even more accurate answer to the following question: How much of the source code has been covered by the unit tests?

```
Coverage = (CT + CF + LC)/(2*B + EL)
```

where

- CT = conditions that have been evaluated to 'true' at least once
- CF = conditions that have been evaluated to 'false' at least once
- LC = covered lines = lines*to*cover - uncovered_lines
- B = total number of conditions
- EL = total number of executable lines (`lines_to_cover`)

Coverage on new code (`new_coverage`)
Identical to Coverage but restricted to new / updated source code.

Line coverage (`line_coverage`)
On a given line of code, Line coverage simply answers the following question: Has this line of code been executed during the execution of the unit tests?. It is the density of covered lines by unit tests:

`Line coverage = LC / EL`
where

- LC = covered lines (`lines_to_cover` - `uncovered_lines`)
- EL = total number of executable lines (`lines_to_cover`)

Line coverage on new code (`new_line_coverage`)
Identical to Line coverage but restricted to new / updated source code.

Line coverage hits (`coverage_line_hits_data`)
List of covered lines.

Lines to cover (`lines_to_cover`)
Number of lines of code which could be covered by unit tests (for example, blank lines or full comments lines are not considered as lines to cover).

Lines to cover on new code (`new_lines_to_cover`)
Identical to Lines to cover but restricted to new / updated source code.

Skipped unit tests (`skipped_tests`)
Number of skipped unit tests.

Uncovered conditions (`uncovered_conditions`)
Number of conditions which are not covered by unit tests.

Uncovered conditions on new code(`new_uncovered_conditions`)
Identical to Uncovered conditions but restricted to new / updated source code.

Uncovered lines (`uncovered_lines`)
Number of lines of code which are not covered by unit tests.

Uncovered lines on new code (`new_uncovered_lines`)
Identical to Uncovered lines but restricted to new / updated source code.

Unit tests (`tests`)
Number of unit tests.

Unit tests duration (`test_execution_time`)
Time required to execute all the unit tests.

Unit test errors (`test_errors`)
Number of unit tests that have failed.

Unit test failures (`test_failures`)
Number of unit tests that have failed with an unexpected exception.

Unit test success density (%) (`test_success_density`)

```
Test success density = (Unit tests - (Unit test errors + Unit test failures)) / Unit tests * 100
```