Design and Software Quality Metrics

Exploring the Relationship between Design Metrics and Software Diagnosability using Machine Learning (2018)

Thomas Dornberger Sofie Kemper

2018-05-10

1 Metrics

1.1 Static Metrics

The following metrics are *static* metrics, i.e., they are calculated based on the source code.

1.1.1 Size and Complexity Information

In cases where size and complexity metrics can only be measured at the method or file level, we aggregate the calculated values such that we obtain the maximum, average and median values for the whole project.

Lines of Code including Comments (LOC) The *number of lines of code* qantifies a component's size by counting the number of lines of code excluding blank lines. Comments are included in this measure. We use TeamScale (see 2.0.2) to measure the number of lines of code.

Lines of Code without Comments (SLOC) The *number of lines of code without comments* or *source lines of code* measures the size of actual source code of a component, i.e., it quantifies the size of the source code excluding blank lines and comments. We measure it using Lizard (see 2.0.1) as well as TeamScale (see 2.0.2).

High Length Methods (F-HML) This metric quantifies the occurrence of high-length methods, i.e., methods with more than 75 SLOC. It is provided by TeamScale (see 2.0.2).

In addition, we measure the proportion of high-length methods (F-PHML).

Medium Length Methods (F-MML) This metric counts the number of methods of medium length. These are methods that have between 30 and 75 SLOC. We use TeamScale (see 2.0.2) to measure it.

In addition, we use the proportion of methods of medium length (F-PMML).

Files of High Size (F-HFS) This metric quantifies the occurrenc of very long files, i.e., files containing more than 750 SLOC. It is extracted using TeamScale (see 2.0.2).

In addition, we calculate the proportion of files of high size (F-PHFS).

Files of Medium Size (F-MFS) Using TeamScale (see 2.0.2), we obtain this number of medium to long files, i.e., files containing between 300 and 750 SLOC.

In addition, we measure the proportion of files of medium size (F-PMFS).

Methods with High Nesting Depth (F-HND) The nesting depth is an indicator for the complexity of a method. It measures the number of statement blocks nested due to the use of control structures. This metric quantifies the occurrence of methods with a very high nesting depth, i.e., a nesting depth higher than 5. It is provided by TeamScale (see 2.0.2).

In addition, we use the proportion of methods with high nesting depth (F-PHND).

Methods with Medium Nesting Depth (F-MND) This metric indicates the number of methods with medium nesting depth, i.e., methods with a corresponding nesting depth between 3 and 5. It is provided by TeamScale (see 2.0.2).

In addition to this absolute number, we calculate the proportion of methods with medium nesting depth (F-PMND).

Token Count of Functions(TCF) The *token count of a function* describes the number of conditional statement tokens in a function. It is used to calculate the cyclomatic complexity number. We use the tool Lizard (see 2.0.1) to measure this function-level property and aggregate the obtained values as described above.

Parameter Count of Functions (PCF) The *parameter count of a function* quanitifies the number of parameters a given function takes. We measure this function-level property using Lizard (see 2.0.1) and aggregate the obtained values as described above.

Cyclomatic Complexity Number (CCN) The *cyclomatic complexity* or *McCabe's complexity* of a component describes its perceived complexity. It is proportionate to the number of linearly independent paths through a program, i.e., if-statements or while-loops increase this value. A high cyclomatic complexity number indicates that code is hard to read, understand, and maintain which might lead to more bugs. We measure the cyclomatic complexity on a method-level using Lizard (see 2.0.1) and aggregate the values as described above. In addition, we also use the cyclomatic complexity numbers (aggregated project-level data) provided by TeamScale (see 2.0.2).

Number of High Cyclomatic Complexity Methods (HCC) This metric counts the number of methods which are scored as highly complex, i.e., those with a cyclomatic complexity number greater than 20. We calculate it using TeamScale (see 2.0.2).

Number of Medium Cyclomatic Complexity Methods (MCC) This metric represents the number of methods of medium complexity, i.e., those with a cyclomatic complexity number between 10 and 20 ($CC \in (10, 20]$). It is provided by TeamScale (see 2.0.2).

Number of Low Cyclomatic Complexity Methods (LCC) This metric is used to measure the number of methods of low complexity. Low complexity methods are those that show a cyclomatic complexity number smaller than 10. TeamScale (see 2.0.2) is used to obtain this metric.

Proportion of High/Medium/Low Complexity Methods (PHCC/PMCC/PLCC) We define the proportions of high (respectively medium or low) complexity methods by using the absolute number of high (respectively medium or low) complexity methods (see above) provided by TeamScale (see 2.0.2) and normalising it by the total number of methods measured, thus, obtaining a relative occurrence of highlyy (respectively moderately or lowly) complex methods.

1.1.2 Coupling and Cohesion Metrics

The following metrics try to capture coupling and cohesion of the source code. They are all provided by the tool JPeek (see 2.0.3). we use the aggregated metrics provided by this tool where applicable: minimum value, maximum value, number of classes scored as "Green", "Yellow", and "Red", respectively. In addition, the total score and the percentage of defects are used for each of the 5 metrics.

Cohesion Among Method of Class (CAMC) This metric measures the cohesion among the methods of a given class. Formally, it indicates the extent of intersections of individual method parameter type lists with the list of parameter types in all methods in the class.

Lack of Cohesion of Methods (LCOM5) This metric was proposed by Henderson and Sellers and can be interpreted as the number of pairs of methods in a given class having no common attribute. Hence, it is based on method similarity among methods of a class. It provides a measure of class cohesion expressed as percentage value ($\in [0, 1]$). A value of 0 indicates full cohesion while a value of 1 indicates no cohesion.

Method-Method through Attributes Cohesion (MMAC) This metric is an indicator of method-method cohesion. The similarity between a pair of methods is expressed as a function of their shared properties. The metric defines the average cohesion of all pairs of methods.

Normalised Hamming Distance (NHD) This metric measures class-level cohesion uses the similarity of parameter types of the class's methods as basis for measuring cohesion. It compares pairs of methods, counting a disagreement only if a parameter type is used by one of the methods and not used by the other method. It was developed as an alternative to CAMC in order to prevent false positives and have a measure with a finer granularity.

Sensitive Class Cohesion Metric (SCOM) This metric measures cohesion via the proportion of class attributes used in the class's methods. It yields values in the range $[0, 1]$ where 0 indicates total lack of cohesion, i.e., every method deals with an independent set of attributes, and 1 indicates full cohesion, i.e., all class attributes are used by every single method.

1.1.3 FindBugs-Findings

All following findings are provided by FindBugs which is integrated in TeamScale (see 2.0.2).

Performance Code Smell Findings (FB-P) This metric counts the number of found code smells regarding performance. Examples include the boing and subsequent unboxing of a value or the invocation of `.toString()` on a String.

Malicious Code Vulnerability Findings (FB-MCV) *Malicious code vulnerabilities* quantifies the number of of code smells in this category, e.g., a `finalize` method that is `public` instead of `protected`.

Security Findings (FB-SEC) This metric measure the number of finding in the code smell category security, e.g., a JSP-reflected cross-site scripting vulnerability.

Dodgy Code Findings (FB-DC) This metric quantifies the occurrence of "dodgy code", e.g., unchecked/unconfirmed casts.

Correctness Findings (FB-COR) *Correctness findings* counts the number of code smell findings in the category of correctness, e.g., impossible downcasts.

Multithreaded Correctness Findings (FB-MCOR) This metric measures the number of findings regarding multithreaded correctness, e.g., attribute with a `get`-method that is not `synchronized` while the corresponding `set`-method is `synchronized`.

Bad Practice Findings (FB-BP) This metric counts the number of bad practice findings, e.g., using a rough value of a known constant.

1.1.4 TeamScale-Findings

All following metrics are obtained via the tool TeamScale (see 2.0.2)

Number of TeamScale-Findings (CF) This metric provides the total, aggregated number of findings by the tool TeamScale (see 2.0.2), i.e., including TeamScale's own as well as FindBug's code findings. All the findings listed below as well as the findings regarding structure (F-HML, F-MML, F-HFS, F-MFS, F-HND, F-MND) are contained in this number.

Missing Braces for Block Statements Findings (F-MBB) This metric quantifies the occurrence of the code anomaly that block statements miss braces.

Null Return Optional Findings (F-RT) This metric counts how often a method returns `null` although the return type is `Optional`.

Missing Code Findings (F-MC) *Missing code findings* quantifies how often empty blocks, code that is "commented out" or files which don't contain any code occur.

Test Convention Findings (F-TC) This metric counts how often test conventions are violated. This includes the naming of test classes as well as the usage of `@ignore` and inverted conditions.

Null Pointer Dereference Findings (F-NP) This metric quantifies the occurrence of possible `null` pointer dereferences at runtime due to wrong `null` assignments or missing checks before dereferencing.

Unused Variable/Parameter Findings (F-UVP) This metric counts the number of unused variables or parameters.

Exception Handling Findings (F-EH) *Exception Handling Findings* represents the number of code smell findings regarding exception handling, e.g., catching or throwing generic exceptions or the loss of the stacktrace.

`Contains` on List Findings (F-CL) This metric quantifies how often `contains()` is called on a list, which is a performance issue.

Bad Practice Findings (F-BP) The category of bad practice code smells includes star imports or methods with the same name as methods in `Object`. They supplement the bad practice findings provided by FindBugs. This metric quantifies the occurrence of such findings.

Unused Code Findings (F-UC) This metric quantifies the occurrence of unused `private` fields or methods.

Code Formatting Findings (F-CF) This metric counts how often problems regarding the code formatting are found. This includes the problems of multiple statements or declarations in the same line.

Cloning Findings (F-CL) The *cloning findings* metric counts the number of clones, i.e., duplicated code. A high number of clones can cause problems regarding the code maintenance and, thus, introduce bugs.

Clone Coverage (F-CLC) The *clone coverage* describes how likely it is that a random SLOC is cloned at another position as percentage.

1.2 Dynamic Metrics

The following metrics are *dynamic* design metrics, i.e., they are calculated based on a program's call or data dependency graph.

1.3 Test Suite Characteristics

The following metrics aim to quantitatiely analyse a project's test suite.

1.4 Bug Characteristics

The following metrics are related to a known bug and aim to quantify features of the bug such as its location and size.

2 Tools

We have tried a multitude of different tools (SourceMeter, LOCC, SonarQube, CCCC, USC CodeCount, CLOC, etc.) and chosen the following for the interesting metrics they provide as well as the possibility to automate their usage, i.e., the possibility to execute them on the command line and output formats that can be easily parsed and used for our purposes.

2.0.1 Lizard

Lizard is a Python-based tool that analyses code size and perceived code complexity as well as parameter and token counts at the function level. It can analyse Java, C/C++, JavaScript, Python, Ruby, Swift, PHP, Scala, and Objective C scripts. More information can be found at `https://pypi.org/project/lizard/`.

2.0.2 TeamScale

TeamScale is a tool that provides a multitude of static code analyses to indicate code quality and possible quality defects ("findings"). It is free for academic usage. The tool monitors the quality of code over time by using manualyy checked-in versions or automatically detected versions of a given repository. By using the diffs between versions, it can quickly analyse the history of a project. The tool FindBugs (see `http://findbugs.sourceforge.net`) is also contained in TeamScale. The metrics are calculated on the file or even method level and aggregated hierarchically. Hence, we can obtain project metrics easily.

More information regarding TeamScale can be found at `https://www.cqse.eu/en/products/teamscale/landing/`.

We have implemented a REST-client to automatically query the metric values from TeamScale and to transform them into a format we can easily use for our analysis.

2.0.3 JPeek

JPeek is a static collector of Java code metrics relating to cohesion and coupling. It measures 5 metrics on the method level and aggregates this data quantitatively, e.g., calculating min, max, variance, and other statistical measures, as well as qualitatively, i.e., "scoring" components as a whole and categorising them into the three classes "green", "yellow", and "red". In addition, it defines and measures "defects" which are classes whose scores particularly bad using the mean scores as baseline.

We use the version JPeek 0.26.3. For more information on the tool, the interested reader is referred to `http://www.jpeek.org` and `https://github.com/yegor256/jpeek`.