Design and Software Quality Metrics

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

Thomas Dornberger Sofie Kemper

2018-05-31

1 Metrics

1.1 Static Metrics

The following metrics are *static* metrics, i.e., they are calculated based on the source code. All metrics that might be correlated to the size of the project's code base are normalised by the number of the lines of source code in the project. This is indicated by the postfix "-D", e.g., F-ML-D.

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 2.2) 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 2.1) as well as TeamScale (see 2.2).

High Length Methods (F-ML) This metric quantifies the occurrence of methods that are of medium to high length, i.e., methods with more than 30 SLOC. It is provided by TeamScale (see 2.2).

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

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 2.2).

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

Files of Medium Size (F-MFS) Using TeamScale (see 2.2), 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 2.2).

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 2.2).

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 2.1) 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 2.1) 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 2.1) and aggregate the values as described above. In addition, we also use the cyclomatic complexity numbers (aggregated project-level data) provided by TeamScale (see 2.2).

Maximum Cyclomatic Complexity (MAXCC) The maximum cyclomatic complexity quantifies the complexity of the most complex file in the analysed source code.

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 2.2).

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 2.2).

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 2.2) 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 2.2) 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 2.3). 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 2.2).

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 2.2)

Number of TeamScale-Findings (CF) This metric provides the total, aggregated number of findings by the tool TeamScale (see 2.2), 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.

Performance: `Contains` on List Findings (F-PCL) 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 as generated by the tool JDCallgraph (see 2.3.1). All metrics are computed on the data dependency graph (prefix DD-) as well as the callgraph (prefix CG-) using the statistical programming language R and the igraph library (see 2.4). We use the multi-edge versions of these graphs and simplify them where appropriate (see below).

1.2.1 Vertex Count (DD-VC, CG-VC)

The vertex count is a measure of graph size. It quantifies the number of vertices in the graph.

1.2.2 Edge Count (DD-EC, CG-EC)

The edge count is a measure of graph size, indicating the number of edges (including multi-edges) between pairs of vertices of the graph.

1.2.3 Simplified Edge Count (DD-SEC, CG-SEC)

The simplified edge count corresponds to the number of edges in the graph, not counting multi-edges, i.e., if there are multiple edges between one pair of vertices, only one of these edges is counted.

1.2.4 Multi-Edge Proportion (DD-MEP, CG-MEP)

This metric indicates the proportion of multi-edges in the graph, i.e., which fraction of the edges present in the graph are actually multi-edges.

1.2.5 Maximum Vertex Degree (DD-MAXVD, CG-MAXVD)

The maximum vertex degree is the maximum total degree of any vertex in the graph.

1.2.6 Mean Vertex Degree (DD-MVD, CG-MVD)

The mean vertex degree is the average of all vertex total degrees in the graph.

1.2.7 Vertex Degree Quantiles (DD-VD*Q, CG-VD*Q)

We look at the vertex degree distribution by aggregating the data in statistical quantiles. The 90-th, 80-th, 75-th and 50-th (median) quantiles are used. For instance, the 90-th quantile (denoted by CG-VD90Q) indicates the threshold below which 90% of all vertex degrees lie.

1.2.8 Maximum Vertex In-Degree (DD-MAXVI, CG-MAXVI)

The maximum vertex in-degree is the maximum in-degree of any vertex in the graph, where in-degrees are the number of directed edges with the given vertex as end point.

1.2.9 Mean Vertex In-Degree (DD-MVI, CG-MVI)

This metric is calculated as the average of all vertices' in-degrees.

1.2.10 Maximum Vertex Out-Degree (DD-MAXVO, CG-MAXVO)

The maximum vertex out-degree indicates the maximum number of outgoing edges for all vertices in the graph.

1.2.11 Mean Vertex Out-Degree (DD-MVO, CG-MVO)

This metric denotes the average of all vertices' out-degrees.

1.2.12 Mean Start-Node Degree (DD-MSND, CG-MSND)

This metric denotes the average degree of the start nodes. Start nodes are the test cases, i.e., those nodes in the graph which possess only outgoing edges. It indicates how many methods are called directly by the test cases.

1.2.13 Graph Diameter (DD-GD, CG-GD)

The graph diameter is a measure of a graph's size, indicating the geatest distance between any pair of vertices.

1.2.14 Graph Radius (DD-GR, CG-GR)

The graph radius denotes the minimum eccentricity of any vertex in the graph. The eccentricity of a vertex indicates how far this vertex is from that vertex most distant from it in the graph.

1.2.15 Mean Distance (DD-MD, CG-MD)

This metric indicates the mean (geodesic) distance between all pairs of nodes in the graph.

1.2.16 Maximum Eigenvector Centrality (DD-MEC, CG-MEC)

A vertex' eigenvector centrality indicates its importance or influence in the graph, e.g., nodes that are connected to many nodes, especially to many other important nodes, show a high eigencentrality. Each vertex has a relative score. Google's PageRank is a randomised version of eigenvector centrality which is more robust.

This metric denotes the maximum eigencentrality found in the graph.

1.2.17 Eigenvector Centrality Quantiles (DD-EC*Q, CG-EC*Q)

This metric characterises the distribution of vertices' eigenvector centrality scores by providing the 90-th, 80-th, 75-th, and 50-th quantiles (see above) of all the eigencentrality scores in the graph.

1.2.18 Vertex Connectivity (DD-VCON, CG-VCON)

The vertex connectivity is a metric that indicates how many vertices need to be removed to render the graph disconnected, i.e., ensure that every vertex can no longer reach every other vertex. It might indicate how dense the graph, and thus, how complex its call- or data-dependency-structure, is.

1.2.19 Edge Connectivity (DD-ECON, CG-ECON)

A graph's edge connectivity indicates how many edges need to be removed from the graph to create a disconnected graph.

1.2.20 (Average) Clustering Coefficient (DD-CC, CG-CC)

A vertex' clustering coefficient captures the local connectivity, i.e., it measure how likely it is that any two of its neighbour's are connected amongst each other. This score is aggregated via averaging over all vertices in the graph to get a clustering coefficient for the entire graph, which indicates the extent to which clusters exist in the graph.

1.3 Test Suite Characteristics

The following metrics aim to quantitatiely analyse a project's test suite. We use basic metrics provided via Gzoltar (see 2.5) as well as test suite metrics proposed by Perez et al in “A Test-suite Diagnosability Metric for Spectrum-based Fault Localization Approaches” and shown to be good indicators of spectrum-based fault localisation accuracy. These metrics are computed based on the coverage data provided by Gzoltar (see 2.5).

On top of that, we use the number of test cases and test size normalised via the number of source lines of code as a coarse approximation of test coverage which would be infeasible to calculate with our limited computation power. Other metrics such as the number of relevant components (i.e., all components tested by at least one of the relevant tests) were rejected since equivalent information is already contained in the dynamic metrics.

1.3.1 Number of Relevant Passed Test Cases (T-NP)

The total number of test cases out of the test suite that were passed, i.e., executed successfully. We measure this based only on the relevant test classes, i.e., all test classes which load any of the changed components. This metric is calculated via Gzoltar (see 2.5).

1.3.2 Proportion of Passed Test Cases (T-PP)

The proportion of passed test cases out of all relevant test cases.

1.3.3 Number of Failed Test Cases (T-NF)

The total number of test cases out of the relevant test suite that were failed, i.e., that could not be executed successfully. This number can be extracted by Gzoltar (see 2.5) or accessed via `http://program-repair.org/defects4j-dissection/#!/`.

1.3.4 Proportion of Failed Test Cases (T-PF)

The proportion of failed test cases out of all relevant test cases. This number can be extracted by Gzoltar (see 2.5) or accessed via `http://program-repair.org/defects4j-dissection/#!/`.

1.3.5 Number of Test Cases (T-N)

This metric counts the total number of test cases in the given test suite.

1.3.6 Number of Relevant Test Cases (T-RN)

The number of test cases that are relevant for the buggy version, i.e., the number of test cases in the classes in which at least one test case fails because of the bug. This measure is provided by gzoltar (see 2.5).

1.3.7 Proportion of Relevant Test Cases (T-RP)

This metric determines which proportion of all test cases in the test suite are considered relevant, i.e., load at least one of the changed methods during execution.

1.3.8 Test Case Density (T-DSLOC)

This metric measures number of test cases per 100 source lines of code, i.e., it gives an indicator of how well-tested the source code is.

1.3.9 Test Size (T-SLOC)

This metric indicates the size of the tests by measuring the total number of lines of code (excluding blank lines and comments) of tests.

1.3.10 Relative Test Size (T-RSLOC)

The relative test size quantifies the size of the test suite (measured via source lines of code) normalised by the size of the project (also in source lines of code) since we expect a strong positive correlation between these two features/

1.3.11 Test Density (T-D)

The test density is a measure that ensures that components are frequently involved in tests. It is computed as T-D= $\sum A[i,j]/(N \cdot M)$ and then normalised, s.t., a value of 1.0 is the target.

1.3.12 Test Diversity (T-G)

The test diversity tries to quantify to what extent components are tested in diverse combinations. We compute it as T-G= $1 - \sum n \cdot (n-1)/(N \cdot (N-1))$ which yields values in the interval $[0;1]$ where 1 is seen as the optimal value.

1.3.13 Test Uniqueness (T-U)

The test uniqueness tries to measure to what extent components are distinguishable based on the test results. It is computed as T-U=|T-G|/T-M.

1.3.14 Perez et al's diagnostic predictor (T-DDU)

This metric tries to combine the previous ones in a way that provides the maximum amount of information concerning the test suite's influence on software diagnosability. It is partly based on the notion of entropy. The metric is computed via the formula T-DDU = T-P $\cdot$ T-G $\cdot$ T-U and, thus, produces values in the interval $[0;1]$ where a value of 1 is regarded as ideal.

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. The following statistics are taken from the Defects4J Dissection (see 2.6).

In addition, for all static metrics analysed via TeamScale on the project level, we define buggy-file versions (prefix BF-) which measure the corresponding statistic only on the files that are responsible for the bug, i.e., those files that are modified in the bug's corresponding patch. The aggregation of these metrics for multiple files is achieved in the same way as the project-level aggregation, i.e., some findings are summed up whereas others are normalised by the respective source lines of code or the maximum values are used (e.g., maximum cyclomatic complexity number).

On top of that, the dynamic metrics we defined on the project level are also extended to the bug level (prefix B-) by looking at bug-specific properties of the call- and data-dependency graphs, e.g., the in-degree of the vertices which correspond to the faulty methods. In cases where there are multiple faulty methods, the "most extreme" value is used, e.g., the maximum in-degree or the minimum eccentricity.

Other ideas such as analysing whether the bug is part of a design antipattern or calculating object-oriented metrics on the bug method were rejected due to their high effort and low presumed meaningfulness for the software diafnosability.

1.4.1 Number of Files (B-NF)

This metric quantifies the number of files modified in the patch that fixes the bug.

1.4.2 Number of Classes (B-NC)

The number of classes changed in the patch is given by this metric.

1.4.3 Number of Methods (B-NM)

This metric indicates the number of methods modified in the patch.

1.4.4 Number of Lines (B-NL)

The metric counts the total number of lines modified in the patch.

1.4.5 Number of Lines Added/Removed/Modified (B-NAL/B-NRL/B-NML)

These metrics indicate the number of lines (of code) that were added, removed, or modified, respectively, in the patch. The sum of these three metrics corresponds to the number of lines (B-NL).

1.4.6 Number of Chunks (B-NCH)

This metric indicates the number of "chunks" modified in the patch. A chunk is defined as a section in the code containing sequential line changes.

1.4.7 Number of Repair Patterns (B-NRP)

This metric quantifies the number of different "repair patterns" employed in the patch for the corresponding bug. A repair pattern is a general pattern of fixing a bug, i.e., a characteristic of the bug patch. Examples include "Single Line", indicating that only a single line was changed, or "Wrong Variable Reference".

1.4.8 Number of Repair Actions (B-NRA)

The number of repair actions counts the number of different actions employed in the process of fixing the bug, i.e., visible in the patch diff. Examples of repair actions include "Variable replacement by another variable" or "Conditional (if) branch addition".

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.

2.1 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/`.

2.2 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.

2.3 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`.

2.3.1 JDCallgraph

JDCallgraph is a tool for dynamic call graph generation for Java. Call graphs as well as data dependency graphs can be computed and are provided in .dot format. In addition, JDCallgraph can provide coverage and other information which we do not use in this project. More details can be found via `https://github.com/dkarv/jdcallgraph`.

The .dot-files are read into R (see 2.4) via the R library sna (see `https://www.rdocumentation.org/packages/sna`).

2.4 R igraph Library

igraph is a library for network analysis. It is available for R, Python, and C and allows fast and comfortable processing of large graphs. Many standard graph metrics as well as an efficient graph data structure are provided by this library. More information can be found via `http://igraph.org/r/`.

2.5 Gzoltar

Gzoltar is a tool used for spectrum-based fault localisation, i.e., for automatic testing and debuging. It is provided as a command-line tool as well as an Eclipse plugin. It produces coverage information as well as suspiciousness scores for different artifacts. More information can be found via `http://gzoltar.com`.

2.6 Defects4J Dissection

The website `http://program-repair.org/defects4j-dissection/#!/` provides data that describe the defects4j bug dataset. For all buggy versions in defects4j it provides summary statistics of the bug. Additionally, the corresponding patch can be accessed.