Software Evolution
Practical Lab Series 1 - Software Metrics

Gerben van der Huizen - 10460748
Vincent Erich - 10384081

November 24, 2016

Introduction

This document briefly describes the methods that we have implemented in order to calculate the SIG Maintainability Model scores for the Java projects/systems listed in the assignment description of Series 1 (`smallsql` and `hsqldb`). Furthermore, this documents also explains the reasoning behind the thresholds chosen for each metric.

Metric 1: Volume

The volume metric value is calculated by counting the number of lines of source code that are not comment or blank lines. The Rascal module `computeVolume` contains all the methods for calculating the volume metric value and rank.

First, the Rascal method `visibleFiles` is used to obtain a set with all the source files of the Java project (the argument given to `visibleFiles` is the path to the source directory of the Java project). Next, the Rascal method `readFileLines` is applied to every source file in order to obtain a list with all the lines in the source file (per source file). This list is then filtered so that it only contains lines that are not comment or blank. Finally, the sizes of the filtered lists of all the source files are summed, and the resulting number is the volume metric value.

The rank of the volume metric is determined by using the ranking scheme provided by [2, p. 34] (for Java). It is important to note that our implementation of the volume metric only allows for comparison between Java projects (i.e., it is not possible to compare the rank of the volume metric for a Java project to the rank of the volume metric for a Cobol project).

Metric 2: Unit Size

The unit size metric value is calculated by determining a risk level for every unit in a project. In case of a Java project, a unit is a method or a constructor. Then, for each risk level, the percentage of lines of code that falls within units categorized at that level is calculated. The Rascal module `computeUnitSize` contains all the methods for calculating the unit size metric value and rank.

First, a M3 model is created of the whole Java project[1]. Next, the constructor locations and method locations are extracted from this model. The Rascal method `readFileLines` is then applied to every unit location in order to obtain a list with all the lines that fall within an unit (constructor or method). This list is filtered so that it only contains lines that are not comment or blank. The size of the filtered list determines the risk level of the unit. The risk level of a unit is determined by using the ranking scheme in table III (a) from [1]. Every unit is assigned a risk level: *low* (LOC ≤ 30), *moderate* ($30 <$ LOC ≤ 44), *high* ($44 <$ LOC ≤ 74), or *very high* (LOC > 74). The number of LOC in each risk level category and the total number of LOC (of all the units) are then used to calculate the unit size metric value (i.e., the percentages described in the previous paragraph).

The rank of the unit size metric is determined by using the ranking scheme in table IV (b) from [1] (where five stars equals a '++' rank and one star equals a '--' rank).

Metric 3: Unit Complexity

The complexity per unit metric value is calculated by determining a risk level for every unit in a project. Then, for each risk level, the percentage of lines of code that falls within units categorized at that level is calculated (such as for the unit size metric value). The Rascal module `computeUnitComplexity` contains all the methods for calculating the complexity per unit metric value and rank.

Similar to computing the unit size metric value, the M3 model is used to extract information about the units of a Java project. Next, the source file locations are extracted from this model. The Rascal method `createAstFromFile` is then applied to every location in order to obtain a list with Abstract Syntax Trees (ASTs) of the units (constructors and methods) in the source file[2]. The unit ASTs are used to calculate the cyclomatic complexities of the units in the source file. The cyclomatic complexity of a unit is calculated by recursively traversing the unit AST and adding one (starting from one) for each node that would generate a fork in the Java control flow graph [3, p. 599-600]. The cyclomatic complexity of a unit determines the risk level of the unit. The risk level of

[1] It is also possible to create a M3 model of every source file in the Java project, but this caused the run time to be significantly slower.

[2] Our initial implementation used the Rascal method `getMethodASTEclipse` to construct an AST of the whole Java project. However, this caused a memory overflow when tested on the large Java project/system (`hsqldb`).

a unit is determined by using the ranking scheme provided by [2, p. 35]. Every unit is assigned a risk level: *low* ($1 \leq \text{CC} \leq 10$), *moderate* ($11 \leq \text{CC} \leq 20$), *high* ($21 \leq \text{CC} \leq 50$), or *very high* (CC *geq* 74). The number of LOC in each risk level category and the total number of LOC (of all the units) are then used to calculate the complexity per unit metric value (i.e., the percentages described in the previous paragraph).

The rank of the complexity per unit metric is determined by using the ranking scheme provided by [2, p. 35].

Metric 4: Duplication

The duplication metric value is calculated by computing the percentage of all code that occurs more than once in equal code block of at least six lines (i.e., duplicated code blocks over six lines). The Rascal module `computeDuplication` contains all the methods for calculating the duplication metric value and rank.

First, the Rascal method `visibleFiles` is used to obtain a set with all the source files of the Java project (the argument given to `visibleFiles` is the path to the source directory of the Java project). Next, a list relation is constructed that contains relations with three elements: (1) a line of source code, (2) a file index (i.e., the index of the source file that the line of code occurs in), and (3) a line index. This list relation is constructed by looping over all the source files of the Java project. Note that only lines of source code that are not comment or blank lines are taken into account. The list relation is then filtered so that it only contains relations with lines that occur in more than one source file (this can be done using set difference). It is then possible to find code blocks of at least six lines that occur more than once by using the file indices and line indices. When comparing code lines, leading spaces and trailing spaces are ignored. Finally, the duplication metric value is calculated as described in the previous paragraph.

The approach described in the previous paragraph is significantly faster than our initial approach. Our initial approach was as follows: First, filter all the source files of the Java project so that they only contain lines of source code that are not comment or blank lines. Next, start with the first line of the first (filtered) source file, create a code block of six consecutive lines, and check whether that code block occurs in any of the other source files (or in the same source file). If not, then take the next code block of six consecutive lines, etc. If so, then incrementally add one line to the code block until no match is found in any of the other source files. If the largest code block is found that also occurs in any of the other source files, then the code block is counted as duplicate code and the loop continues. This approach seemed to be very accurate, but turned out to be very inefficient: on the small Java project/system (`smallsql`) it took more than two minutes to compute the duplication metric value. The implemented approach is significantly faster than the initial approach, but seems to be a little less accurate since it does not take consecutive lines of code into account, but detects whether each line in a code block of at least six lines also

occurs in any of the other source files.

The rank of the duplication metric value is determined by using the ranking scheme provided by [2, p. 36].

References

[1] Alves, T. L., Correia, J. P., and Visser, J. (2011). Benchmark-based aggregation of metrics to ratings. In *Software Measurement, 2011 Joint Conference of the 21st Int'l Workshop on and 6th Int'l Conference on Software Process and Product Measurement (IWSM-MENSURA)*, pages 20–29. IEEE.

[2] Heitlager, I., Kuipers, T., and Visser, J. (2007). A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE.

[3] Landman, D., Serebrenik, A., Bouwers, E., and Vinju, J. J. (2016). Empirical analysis of the relationship between cc and sloc in a large corpus of java methods and c functions. *Journal of Software: Evolution and Process*, 28(7):589–618. JSME-15-0028.R1.