UNIVERSITY OF AMSTERDAM

SERIES 1

Calculating software metrics

November 20, 2022

Student:
Quinten van der Post, Jorrit Stutterheim
10272380, 13957899

Affiliation:
Master Software Engineering

Lecturer:
Thomas van Binsbergen

Course:
Software Evolution

Contents

1 Introduction

In this report we discuss static code analysis through the calculation of metrics of quality and the creation of maintainability profiles. In the *Methodology* section we shortly describe how each metric calculation has been implemented, and how an associated risk profile for that metric was calculated. In the *results* section we show the scores and maintainability rating of two java projects and discuss the results. Finally in the *discussion* section we discuss caveats of our implementations and areas where improvements can be made.

2 Methodology

2.1 Volume

2.1.1 Line of code calculation

To calculate the amount of lines of code we decided to make a small island grammar that explicitly filters all comments and white space. The grammar simply creates two nodes *'Comments'* and *'NonComments'* this node always end at the EOL. To count all lines of code we simply count all 'NonComment' nodes. This makes counting LoC trivial and has some added advantages in catching strange edge-cases by design, see two examples in the code below.

```
public void main(String [] args) {
    // Parsed as a single NonComment token, thus counted as 1 LoC.
    SomeClass x = /*SomeComment*/ new SomeClass();

    // Parsed as a Comment token and NonComment token, thus
    // counted as 1 LoC.
    /* Some comment */ System.out.println("Starting␣main");

    // Parsed as two NonComment tokens, thus counted as 2 LoC.
    SomeClass y = /*SomeComment*/
    new SomeClass();

}
```

The grammar was written by us, but the 'comments' syntax was borrowed from the Rascal M3 AST. This is noted in the source code as well.

2.1.2 Risk profile

Based on the 'function point' table from [4] we calculated that on average a Java programmer will produce approximately 9540 LoC per year. Divide total project LoC with 9540 to get the total amount of 'man years (MY)' that the project will cost to make. This volume profile can then be graded by the below schema [2].

Grading	M. Y.	Approx. KLoC
++	0-8	0-66
+	8-30	66-246
o	30-80	246-665
-	80-160	655-1,310
–	>160	>1, 310

Table 1: Volume ranking

2.2 Unit Size

2.2.1 Calculating unit size

For calculating unit-size we simply 'visit' all 'methods' in the AST, this returns the location of the method. Using this location (Rascal allows us to read a snippet of code from source) we calculate the lines of code within the method by using our previously mentioned island grammar. This all results in a hash-map with the location of the method as key and the lines of code as value.

2.2.2 Risk profile

To give a rating to unit size create four risk profile categories, based on the maintainability model [2]. The categories are depicted in the following table with the threshold of LoC to end up in that category per method.

Category	Method LoC
Low	<= 20
Moderate	<= 30
High	<= 40
Very high	> 40

Table 2: Unit size thresholds

The thresholds are based on a C coding standard from a University of Amsterdam course [5]. In this standard it states a function should be approximately 20 lines of code, but definitely not exceed 30 LoC. For Java specifically there are no official guidelines that mention a ideal method length, thus this classification seemed good enough for our purposes. To then give an actual ranking based on these categories we then used the risk profile given in the maintainability model [2], see the below table.

Rank	Moderate	High	Very high
++	25%	0%	0%
+	30%	5%	0%
o	40%	10%	0%
-	50%	15%	5%
–	-	-	-

Table 3: Ranking table

2.3 Unit complexity

2.3.1 Calculating cyclomatic complexity

Similarly to calculating unit-size we grab all method locations from the AST. These methods are then immediately parsed in a second 'visit' where the cyclomatic complexity of the method will be calculated. This again results in a hash-map with the location of the method as key and the cyclomatic complexity as value.

The cyclomatic complexity is calculated as follows. Initial value of the method is 1. All type of statements in the list below are checked and the cyclomatic complexity is increased by 1 for each hit.

- return
- if
- case
- defaultCase
- foreach
- for
- while
- do
- infix

- conditional
- catch
- throw

2.3.2 Risk profile

Table four has been taken from the maintainability model [2].

CC	Risk evaluation
1-10	simple, without much risk
11-20	more complex, moderate risk
21-50	high risk untestable
> 50	very high risk

Table 4: Cyclomatic Complexity thresholds [2]

To get a risk profile from this information the LoC of each method have been added to each category. Now divide each category LoC with the total LoC to calculate a rating scheme. The same rating scheme from table 3 is used for this.

2.4 Duplication

Duplication is measured as each line of code in a block of 6 lines of code that occurs in a textually identical (sans comments and white space in between lines) block of code. To find those duplicate blocks we first map each overlapping block of 6 lines, e.g. lines 1-6, and 2-7, etc. are considered and mapped to a location that covers each of those blocks. In this map each collision on the key represents a duplicate, and as such we can add the location of that block to a set of locations on the value in the map.

Having found duplicate blocks, we can invert this mapping and explode each of the sets of locations. This gives us a map of each unique location and the block of duplicate code it belongs to. This map can then be restructured based on the file of each location and then by sorting each location within those files based on whether they start before one another. This mapping; of file path to a map of location to code block, allows us to merge overlapping blocks into the largest block of consecutive duplicate lines.

Then we only need to count the lines of code in each of those blocks and we know the number of duplicated lines of code. Though we could also take a shortcut earlier and just counted made a big set of locations from the initial duplicate blocks, this approach sets us up to provide a better analysis on the most egregious duplication in the project and provides insides in potential source code hazards.

2.4.1 Duplication Profile

Table five, from the SIG maintainability model, describes the ranks assigned to projects containing a percentage of its code in duplicates. Each duplicate line is considered in this profile as a percentage of the total number of lines of code.

Rank	Duplication
++	0% - 3%
+	3% - 5%
o	5% - 10%
-	10% - 20%
−	20% - 100%

Table 5: Duplication relative to LoC [2]

2.5 Unit testing

2.5.1 Calculating coverage

To get an idea of how well a project is tested we implemented a simplified testing coverage calculator. We first map all methods in the project by traversing the AST and mapping each method declaration (not found inside *'test files'*) by its origin and name. Afterwards we specifically traverse the AST and count method calls done inside of Test files that find their original type in a source file outside of the *'test files'*.

Ideally we would look at the declaration of the type that a method call is done from, but the AST proved to be unreliable and has a lot of unresolved declaration.

This means that our metric can only be used as an estimate and that we had to build in some safeguards that means we skip any method call for which we cannot find its origin.

By itself this type of coverage does not say a whole lot. It is easy to create 'fake coverage' with test functions that call methods but don't actually test anything. For this we added an extra metric; we count the number of 'assert' statements and compare that to the amount of methods that are being tested. This at least gives an indication of whether the methods that are being tested are actually being tested well.

2.5.2 Risk profile

The coverage risk-profile is calculated based on table 6 from the maintainability model [2]. The assert risk-profile is based on table 7, the percentage is based on the amount asserts vs. the amount of function tested (coverage). Although these numbers are a bit arbitrary there should be at least as many as asserts as methods under test. This is to give an idea whether the test methods actually do test.

Rank	Unit test coverage
++	95% - 100%
+	80% - 95%
o	60% - 80%
-	20% - 60%
$-$	0% - 20%

Table 6: Unit test coverage [2]

Rank	Assert percentage
++	>200%
+	>150%
o	>100%
-	>90%
$-$	>0%

Table 7: Assert rating

2.6 Maintainability score

To calculate the complete maintainability score each individual rating is given a weight as seen in table 8. All scores are then aggregated according to the maintainability table from the paper [2] to give the final scores discussed in section 3. Note, the 'Assert' ranking is not taken into the stability rating at this moment because it has to big of an influence in the current setup.

Rank	Unit test coverage
++	2
+	1
o	0
-	-1
–	-2

Table 8: Profile weight

3 Results

3.1 Small SQL project

See the smaller SQL project metrics and scores in the tables below. Only for **Analysability** the project score reasonable. This is due to reasonable code size balancing out the lack of unit tests. Looking at the data, overall maintainability can probably best be improved by reducing the complexity of the largest methods, this can improve scores all around.

Type	Rank	Notes
Volume	++	LoC: 24.050
Unit Size	−	high: 1.77 % moderate: 5.13 %, very high: 3.01 %, low: 90.09%
Complexity per unit	− −	high: 8.02%, moderate: 7.42%, very high: 10.40%, low: 58.46%
Duplication	−	Dups: 2559; 10.64%
Unit testing	− −	Coverage: 12.30%, Nr. of methods called from tests: 252
Asserts	++	Nr. of assert statements: 961

Table 9: Small SQL metrics

Type	Rank
Analysability	o
Changeability	−
Stability	− −
Testability	−

Table 10: Small SQL maintainability model

3.2 Large SQL project

Type	Rank	Notes
Volume	+	LoC: 172.452
Unit Size	− −	high: 3.50% moderate: 6.23%, very high: 6.84%, low: 83.44%
Complexity per unit	− −	high: 11.46%, moderate: 13.83%, very high: 10.13%, low: 48.11%
Duplication	− −	Dups: 56191; 32.58%
Unit testing	− −	Coverage: 5.92%, Nr. of methods called from tests: 522
Asserts	++	Nr. of assert statements: 693

Table 11: Large SQL metrics

Type	Rank
Analysability	−
Changeability	− −
Stability	− −
Testability	− −

Table 12: Large SQL maintainability model

4 Discussion

Creating a special grammar for counting LoC was a great decision for this project. This in itself makes a lot of the metric calculation easier to do. Most improvements in the current project would be around the *Stability* and *Testability* metrics who are both reliant on our unit-testing metric. The unit testing metric, our simplified coverage is not extremely accurate. And although we do count and print asserts, these are not actually aggregated into the final scores since we didn't make validate the weights of these metrics yet.

4.1 Magic Numbers

As an additional metric, not discussed in the SIG Maintainability model, we consider the use of magic numbers, and literals of any kind for that matter, to form a potential hazard to the long term maintainability of a project. Several researchers indicate that the use literals outside of meaningful binding leads to code that is less readable and more difficult to maintain [3] [1]. Attaching a rank to the use of magic numbers may allow programmers to recognize risks the take for their future selves. What if a magic number is described in a comment, but the number itself is later changed to suit a different need? Now a comment describes a non-existent piece of code, or worse, it actively confuses programmers who do not know the origin or use of the number. What if a number is repeated many times but not changed everywhere when it needs to be altered? There are many cases where a descriptive variable name for either a hardcoded number, or a string literal can prevent later mishaps.

Of course some numbers would be exempt from this profile. Numbers such as 0, 1, 2, -1 are so prevalent and meaningful, they usually do not cause any confusion. The issue lies in domain specific numbers, such as 1024 (when working with Byte chunks), or even 26 (the length of the alphabet). Are they harmful, or harmless under the right scenario?

References

[1] David Bowes et al. "How good are my tests?" In: *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*. IEEE. 2017, pp. 9–14.

[2] Ilja Heitlager, Tobias Kuipers, and Joost Visser. "A Practical Model for Measuring Maintainability". In: *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*. 2007, pp. 30–39. DOI: `10.1109/QUATIC.2007.8`.

[3] Jagadeesh Nandigam, Venkat Gudivada, and Abdelwahab Hamou-Lhadj. "Learning software engineering principles using open source software". In: Nov. 2008, S3H–18. DOI: `10.1109/FIE.2008.4720643`.

[4] *Programming Languages Table*. Mar. 1996. URL: `http://www.cs.bsu.edu/homepages/dmz/cs697/langtbl.htm`.

[5] Steven de Rooij. "C Coding Standard Datastructuren". In: (June 2021).