Software Evolution
Series 1

Rasha Daoud - 11607394
Ighmelene Marlin - 10296050

November 24, 2017

1 Introduction

This report presents the results of assignment – series1. In this assignment we are computing software metrics following the SIG model suggested in [1]. Calculations are done on two provided open-source Java projects.

Our program calculates scores for the suggested maintainability aspects: analysability, changeability, testability and stability. These scores are then used to calculate an overall maintainability score according to Table 1.

To obtain the above, our program measures the following for each of the Java projects:

- Volume: the total number of lines of code for the project,
- Duplication: percentage of the code blocks of at least 6 lines of code that is an exact copy of another piece of code,
- Unit size: the number of lines of code per method,
- Unit complexity: the cyclomatic complexity of each method,
- Test coverage: the percentage of methods being called by unit tests
- Test quality: the number of asserts per method.

properties / characteristics	volume	complexity per unit	duplication	unit size	unit testing
analysability	x		x	x	x
changeability		x	x		
stability					x
testability		x		x	x

Table 1: SIG maintainabilty scores

2 Metrics

2.1 Volume

After discarding blank lines, single-line comments and multi-line comments from the project, the volume is calculated by counting the remaining lines. The score is determined according to the table in Table 2 found in [1].

rank	KLOC
++	0 - 66
+	66 - 246
o	246 - 665
-	665 - 1,310
–	> 1,310

Table 2: Volume ranking for Java code

2.2 Unit Size

The unit size is calculated for each method in the source code. The program calculates LOC after discarding blank lines, single-line comments and multi-line comments.

Once we have the size of each method, we calculate what percentage of the code falls within a certain range (low, moderate, high or very high) using Table 3 found in [2] and use the percentages in Table 5 found in [1] determine the unit size score.

unit size	risk level
0 - 30	low
30 - 44	moderate
44 - 74	high
> 74	very high

Table 3: Unit size aggregation

2.3 Unit Complexity

The program calculates the cyclomatic complexity (cc) based on the number of independent paths in a method. Each fork point in the method adds 1 to the score.

Using an AST of the Java code provided by a Rascal parser, we use visit with pattern matching to detect and count the statements of interest; each case and catch, do, for, foreach and while loops, if statements, and (&&) and or (||) operators and the conditional (?).

Once we have the cc of each method, we calculate what percentage of the code falls within a certain range (low, moderate, high or very high) using Table 4 found in [2] and use the percentages in Table 5 found in [1] determine the unit complexity score.

cc	risk level
0 - 6	low
6 - 8	moderate
8 - 14	high
> 14	very high

Table 4: Unit complexity aggregation

rank	moderate	high	very high
++	25%	0%	0%
+	30%	5%	0%
o	40%	10%	0%
-	50%	15%	5%
–	-	-	-

Table 5: Unit size and unit complexity ranking

2.4 Duplication

The program detects duplicated lines based on the 6-lines block described in the SIG model.

For detecting duplications, we read the cleaned code and extract blocks per file. We use the distribution method which returns a map of the code blocks and their frequency.

Blocks which occur two or more times are considered to be duplicates. For these blocks, the filename and line numbers they cover are inserted into a set. We use a set so that no filename and line number combination do not appear more than once. The number of duplicate lines is the number of elements in the set.

In order to calculate the percentage of duplication we calculate the volume again, but this time we also remove the accolades and leading and trailing white spaces. A score is determined using the table in Table 6 found in [1].

rank	KLOC
++	0 - 3%
+	3 - 5%
o	5 - 10%
-	10 - 20%
–	20 - 100%

Table 6: Duplication ranking

2.4.1 Method

In addition to removing the blank lines, single-line comments and multi-line comments from the code, we also chose to remove the curly braces { and } because while testing on a small case, the additional duplicates found were only of braces. Aside from that, some developers choose to put the brace at the end of the line while another opts for the following line.

We consider the following to be the same:

```
public void greet() {
  System.out.println("Hello, world!");
}
```

```
public void greet()
{
  System.out.println("Hello, world!");
}
```

We also replaced multiple white spaces with a single one and consider the following to be the same:

```
public int getScore() {
  return this.score;
}
```

```
public int getScore() {
  return    this.score;
}
```

For HSQLDB, replacing the multiple spaces returned more duplicates but not enough to affect the ranking; 16.759 without replacing vs. 17.167 with replacing.

2.4.2 Test properties

1. Copying the input will return 100% duplication because all content has been doubled
2. Deduplicating the input will return 0% duplication because no content has be doubled
3. Each duplicated block should return more than one unique location (file + line number)
4. Each location should be counted no more than once.

2.5 Test Coverage

Tests coverage is calculated after extracting all unit test methods. The program also extracts all methods that can be tested in the open-source project.

Using methodInvocation on the model created from the Java project, the program checks whether the caller method is a testing method and the called method is one of the methods to be tested and counts these. The percentage of called methods out of the methods to be tested is returned. And based on Table 7 found in [1], the score is determined.

rank	unit test coverage
++	95 - 100%
+	80 - 95%
o	60 - 80%
-	20 - 60%
–	0 - 20%

Table 7: Test coverage ranking

2.6 Test Quality

Next to the test coverage as a measure for unit testing, the program counts assert statements in the test units as a better measure for a good quality unit tests.

We loop through all classes that extend the test-case class in Java JUnit package. We get the AST of the class and we count assert statements by visiting the implementation of all methods and look for assert statements or any method call that has an assert statement itself.

3 Results

SIG scores for each of the main maintainability aspects is calculated based on combining metrics described in previous section according to Table 1. Table 8 shows the result of calculated metrics and SIG score for both provided projects.

Volume:	24050
Rating:	++
Unit size:	
low:	60%
moderate:	7%
high:	11%
very high:	11%
Rating:	--
Complexity:	
low:	58%
moderate:	4%
high:	7%
very high:	19%
Rating:	--
Duplication:	8%
Rating:	o
Unit-testing:	16%
Rating:	--
Assert-count:	898
Analysability:	o
Changeability:	-
Testability:	--
Stability:	--
Maintainability:	-

Volume:	169184
Rating:	+
Unit size:	
low:	44%
moderate:	10%
high:	12%
very high:	23%
Rating:	--
Complexity:	
low:	45%
moderate:	7%
high:	12%
very high:	25%
Rating:	--
Duplication:	12%
Rating:	-
Unit-testing:	2%
Rating:	--
Assert-count:	188
Analysability:	-
Changeability:	-
Testability:	--
Stability:	--
Maintainability:	-

Table 8: Results of analysis of the two projects. On the left side: SmallSQL. On the right side: HSQLDB.

References

[1] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pp. 30–39, IEEE, 2007.

[2] T. L. Alves, J. P. Correia, and J. Visser, "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)*, pp. 20–29, IEEE, 2011.