Software evolution, series 1

Michael Peters, 12046515 Ewoud Bouman, 10002578

November 25, 2018

1 Introduction

This is our report for the series 1 assignment in the software evolution course. We will introduce the metrics we used to analyze and evaluate the maintainability of the given Java projects. While doing that we we will also describe how we used Rascal to implement them. Finally we list all the results gathered by our analysis.

2 Metrics

This section contains the five metrics we used for our assignment. For each metric we describe how have implemented them and how we have used them for the maintainability analysis.

2.1 Volume metric

According to Heitlager et al., 2007, measuring volume is done with an assessment of the application's LoC (lines of code). Before doing so, all documentation, comments and empty lines are deleted. The result can then be rated while taking the Java language into account.

The implementation starts with the `M3` object for the Java project being measured. Afterwards the `files(M3)` built in method is used to get a list of all files in the project. These are iterated, counted and finally summed to form the total SLoC.

Measuring every individual file is done by first reading the file using `readFile(loc)` to get the content of the file as a `str`. After doing so these strings are cleaned from empty lines and comments by using the `visit` feature. This is all done by using a single regex: `/(\/\/.*)|(\/\*(?:.|[\n\r])*?\*\/)/`. Finally we split the string at every `\n` and filter the result for strings without content. This is done with a regex as well: `/^[\s\t]*$/`.

We use the SLoC to derive the volume metric based on the thresholds of table 1.

Rank	KLOC
++	0-66
+	66-246
o	246-665
-	665-1310
--	>1310

Table 1: Volume ranking scheme for Java systems (Heitlager et al., 2007).

2.2 Unit size metric

For the unit size measure it's required to measure the LoC of all units and group these. A unit is described by Heitlager et al., 2007 as the minimal section of code that can be executed individually.

Taking the `M3` again it is possible to get all methods in the project by using `methods(M3)`. After doing so, each method's SLoC can be counted in the same way as described in 2.1.

For evaluating each method we used the thresholds from table 2 from Magiel Bruntink's presentation (Bruntink, 2018). While evaluating the SLoC of each method we pair these evaluations together with the percentage of the program's SLoC this method represents. This results in a pair like `<small,0.3%>` for example.

SLoC	Evaluation
1-15	Small
16-30	Moderate
30-60	Large
>60	Very large

Table 2: Unit size evaluation mapping based on Magiel Bruntink's presentation (Bruntink, 2018)

After creating a list of these pairs, we group every evaluation and sum up the percentages. At this point we know for example that small methods make up for an `x` percentage of the program. This data can then be mapped to a rating by using the mapping found in table 3.

Rank	Moderate	Large	Very large
++	15%	5%	0%
+	20%	15%	5%
o	30%	20%	5%
-	40%	25%	10%
--	-	-	-

Table 3: Unit size rating mapping based on the 4 star target from Magiel Bruntink's presentation Bruntink, 2018 and taking the same sort of steps as the CC rating in Heitlager et al., 2007.

2.3 Duplication metric

According to Heitlager et al., 2007 the duplication measure must be measured by blocks of 6 lines. So when two blocks of 6 lines are equal we mark these lines as being duplicate.

We implemented this measure by again taking the project's `M3` instance and extracting it's methods using `methods(M3)`. These will be stripped from comments and white spaces again using the same technique as described in 2.1. Afterwards, the code is iterated and all possible blocks of 6 are generated for later use. These blocks are gathered in a `list` of `list[tuple[loc,int,str]]`. This contains all of the blocks with their lines grouped with a unique aspect of each line (file location and index). Good to know is that whenever these lines are compared with each other we make sure they are trimmed and thus trailing white spaces are removed. This is useful when the duplicates must be counted since a set can then be used and no duplicate duplicates will ever be a problem.

After the block generation we iterate all blocks and keep a map of the searched blocks. If we find duplicates, the block's lines are added to the set of duplicates. Finally the percentage of duplicates can be measured by using the measured volume from 2.1 and a rating can be done using table 2.3.

Rank	Duplication
++	0-3%
+	3-5%
o	5-10%
-	10-20%
--	>20%

Table 4: Duplication ranking. (Heitlager et al., 2007)

2.4 Complexity per unit metric

For this metric we determine the cyclomatic complexity on a unit level. This is a representation of the number of different paths through the code. These can be counted by using the following statements and expressions as suggested by (Landman et al., n.d.): if, for, do, while, case, catch, throw, foreach, &&, || and conditional. These must be counted, starting from one, since there is always one path to take. As a unit in Java we take all methods and constructors.

For each file in the project using `files(M3)` we create the Rascal AST using `createAstFromFile(loc, false)`. These AST's can then be visited and by using pattern matching we can get all methods and constructors. Traversing these further enables us to pattern match all relevant statements e.g. `\if(_,_)` for if statements. The total count is paired with the method's SLoC for later use.

At this point we have all methods as pair of CC and SLoC. We can rate this in the same manner described in 2.2 but using different thresholds. The thresholds we use for the complexity per unit metric are found in tables 5 and 6.

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

Table 5: Unit complexity evaluation (Heitlager et al., 2007).

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

Table 6: Complexity per unit rating mapping (Heitlager et al., 2007).

2.5 Maintainability

The end goal of performing all previous measurements on various metrics is to get a rating on how maintainable the code base is. This maintainability attribute is divided into sub attributes: analysability, changeability, stability and testability. The average of these attributes will yield the maintainability score.

In order to measure every sub attribute, we used the mapping table 7. We weighted every source code property equally and afterwards took an average of the sub attributes to get the maintainability rating.

	Volume	Complexity per unit	Duplication	Unit size
Analysability	x		x	x
Changeability		x	x	
Testability		x		x

Table 7: Maintainability mapping based on (Heitlager et al., 2007).

2.6 Bonus: unit interfacing

This metric evaluates the number of arguments within a method. A large number of parameters indicates a code smell meaning that there may be a deeper issue within the code. It could mean that the purpose of the method is badly designed and needs to be restructured in a more logical manner.
Another issue is that a large number of parameters can limit the reuse-ability of the code and hinders the test-ability. A mapping to a rating for this metric can be found in table 2.6.

Rank	Low [0,2)	Moderate [3]	High [4]	Very high [5,∞)
++	-	12.1	5.4	2.2
+	-	14.9	7.2	3.1
o	-	17.7	10.2	4.8
-	-	25.2	15.3	7.1
--	-	-	-	-

Table 8: Thresholds for rating the unit interfacing metric (Alves et al., 2011)
.

3 results

This section contains the results found in the Java projects that have been analysed using our Rascal tool.

3.1 Small SQL

```
Volume metric:
Total lines of java code:             24016
Volume rating based on lines of code: ++
————————
Complexity metric:
Complexity groups:
Low risk percentage:         74.22812426124800
Moderate risk percentage:    8.0531183303900
High risk percentage:        11.766658762200
Very High risk percentage:   5.952098648300
Complexity rating: − −
————————
Unit size metric:
Unit size groups:
Small sized percentage:              46.01849656361800
Moderate sized percentage:           21.9160540672200
Large sized percentage:              16.367085605500
Very large sized risk percentage:    15.698363765800
Unit size rating: −−
————————
Duplicates metric:
Found duplicate lines:          2106
Total amount of lines:         24016
Duplication percentage:        8.769153897000
Duplicates rating: o
————————
Overal scores:
Analysability: o
Changeability: −
Testability:   −−
Maintainability: −
Analysis took: 42.892198 seconds
Bonus metrics:
————————
Unit interfacing metric:
Unit Interfacing groups:
Low risk percentage:               0.92811296534017972
Moderate risk percentage:          0.04920838682071031
High risk percentage:              0.01668806161745828
Very high risk percentage:         0.00599058622165169
Unit interfacing: ++
```

3.2 Hsqldb

```
Volume metric:

Total lines of java code:            168926
Volume rating based on lines of code: +
————
Complexity metric:
Complexity groups:
Low risk percentage:          65.030246321263500
Moderate risk percentage:     14.0470761891100
High risk percentage:         11.3954162660100
Very High risk percentage:    9.5272612251400
Complexity rating: − −
————
Unit size metric:
Unit size groups:
Small sized percentage:              30.225232943843500
Moderate sized percentage:           19.1111931157100
Large sized percentage:              18.6163132436700
Very large sized risk percentage:    32.0472606983000
Unit size rating: − −
————
Duplicates metric:
Found duplicate lines:          23278
Total amount of lines:          168926
Duplication percentage:         13.77999834000
Duplicates rating: −
————
Overall scores:
Analysability: −
Changeability: −
Testability: − −
Maintainability: −
Analysis took: 411.298524 seconds
Bonus metrics:
————
Unit interfacing metric:
Unit Interfacing groups:
Low risk percentage:              0.88181291070738306
Moderate risk percentage:         0.07218786238886742
High risk percentage:             0.026188635485117896
Very high risk percentage:        0.01981059141863162
Unit interfacing: ++
```

4 Bonus: Testing

In order to test every measurement we implemented in our Rascal tool, we included a set of tests. We implemented two types of test.

The first type checks if our rascal implementation parses the files correctly. For example for the volume metric we count the number of code lines where we filter out the comments. To check for comments we are using regular expressions. One test checks if our implementation parses a java file correctly.
Another test checks if our complexity implementation handles all events correctly. For this we created a test scenario listing all possible java control flow statements. Here we test if they are all detected and correctly summed.

The second type of tests checks the helper functions in our implementation. For example if we parse a string correctly or if we return the correct evaluation for a metric.

All these can be run by using `:test` after importing the desired test module.

References

Alves, T. L., J. P. Correia, and J. Visser (2011). "Benchmark-Based Aggregation of Metrics to Ratings". In: *2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, pp. 20–29. DOI: `10.1109/IWSM-MENSURA.2011.15`.

Bruntink, Magiel (2018). "Software Maintainability – Measurement in Practice". University of Amsterdam Lecture.

Heitlager, Ilja, Tobias Kuipers, and Joost Visser (2007). "A Practical Model for Measuring Maintainability". In: *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*. QUATIC '07. Washington, DC, USA: IEEE Computer Society, pp. 30–39. ISBN: 0-7695-2948-8. DOI: `10.1109/QUATIC.2007.7`. URL: `https://doi.org/10.1109/QUATIC.2007.7`.

Landman, Davy et al. (n.d.). "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions". In: *Journal of Software: Evolution and Process* 28.7, pp. 589–618. DOI: `10.1002/smr.1760`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1760`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1760`.