Software evolution, series 1

Michael Peters, 12046515 Ewoud Bouman, 10002578

November 27, 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 we have implemented them and how we have used them for the maintainability analysis.

2.1 Volume metric

The volume metric ranks the maintainability based on the total volume of the system. The idea is that in general a larger system requires more resources to maintain (Heitlager et al., 2007).

Measuring of the volume is done with an assessment of the lines of code metric (LoC). The LoC represents the total number of code lines in a system where at least one character in a line is not part of a comment.

We used the LoC 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).

Our rascal implementation created an `M3` object of the Java project being measured. The `files(M3)` built in method was used to get a list of all files in the project.

Each file was parsed using the `readFile(loc)` to get the content of the file as a `str`. Using a regular expression, each string was parsed to remove empty lines and comments using the `visit` feature. Finally we split the string at every `\n` and counted the number of lines.

2.2 Unit size metric

A code unit represents the minimal section of code that can be executed individually. Larger units can be more difficult to test and maintain, thus lowers the overall maintainability.

To measure unit size as a metric, it's required to measure the LoC of all units and group these in a set of intervals. For each unit in the system we measure the LoC and rank them using the thresholds from table 2 from Magiel Bruntink's presentation (Bruntink, 2018).

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

LoC	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 grouped every evaluation and summed up the percentages. These percentages where evaluated to a rating by the mapping shown in table 3.

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

Table 3: Mapping based on the 4 star target from Bruntink's presentation (Bruntink, 2018) using the same steps as the CC rating in (Heitlager et al., 2007).

In rascal we implemented this metric by taking the `M3` to retrieve all methods in the project using `methods(M3)`. Each method's LoC could then be counted using the approach described in 2.1.

2.3 Duplication metric

Excessive code duplication can be detrimental to the maintainability of a system because it lowers the analysability and changeability (Kapser et al., 2008).

We measured the duplication metric by grouping successive lines of code in blocks of six lines each. All non-overlapping lines that occur-ed more than once in equal code blocks where marked as duplicate. By dividing the number of duplicate lines by the LoC we derived a percentage we ranked using table 2.3.

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

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

We implemented this measure in rascal by taking the project's `M3` instance and extracted it's methods using `methods(M3)`. String lines where stripped using the same technique as described in 2.1 and for this metric we also removed leading white spaces.
The code blocks of the volume where formed and are gathered in a `list` of `list[tuple[loc,int,str]]`. This contained all the blocks with their lines grouped with a unique identifier of each line (file location and index).

Then we iterated over all the blocks and kept a map of the searched blocks. In the event of matching blocks the lines of the block where added to the set of duplicates.

2.4 Complexity per unit metric

The complexity of a unit describes the number of different paths present within a unit. Unit complexity has a negative impact on the analysability and testability of a system because they are more difficult to understand (Heitlager et al., 2007). We counted the number of control flow statements per unit and categorized them based on the thresholds in table 5 to evaluate the complexity.

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).

This categorization is ranked in a similar manner as described in 2.2 using table 6 resulting in the unit complexity metric.

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).

Our rascal implementation parses the following control flow statements as suggested by (Landman et al., n.d.): if, for, do, while, case, catch, throw, foreach, &&, || and conditional. Starting from one, since there is always one path to take, we count all different paths in the unit. As a unit in Java we take all methods and constructors.

For each file in the project using `files(M3)` we created the Rascal AST using `createAstFromFile(loc, false)`. We traversed the AST's and by using pattern matching we retrieved all methods and constructors. Traversing these further enabled us to pattern match all relevant statements e.g. `\if(_,_)` for if statements. We then paired the total count unit's LoC for later use.

2.5 Maintainability

The end goal of retrieving all metrics was to evaluate the maintainability of the system. We divided the maintainability into sub attributes: analysability, changeability, stability and test-ability. An average of these sub attributes was used to derive the maintainability score.

In order to evaluate every sub attribute, we used the mapping shown in 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 was badly designed and should be restructured in a more logical manner.

Another issue is that a large number of parameters could limit the reuse-ability of the code and limit the test-ability. We used the mapping in table 2.6 to rank this metric.

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)

.

We implemented this metric by counting the number of parameters of each method in the project.

3 results

This section contains the results from our analysis on the Java projects 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
_______
Overall 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.

Kapser, Cory J. and Michael W. Godfrey (2008). ""Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software". In: *Empirical Softw. Engg.* 13.6, pp. 645–692. ISSN: 1382-3256. DOI: 10.1007/s10664-008-9076-6. URL: http://dx.doi.org/10.1007/s10664-008-9076-6.

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.