Software evolution, series 1

Michael Peters, 12046515
Ewoud (Johan) Bouman, 10002578

November 21, 2018

1 introduction

This is a report about the series 1 assignment in the software evolution course. In this report will be discussed what measurements we had to perform on the given Java projects and why it contributes to the maintainability attribute. We will also briefly describe how we decided to measure these and how we used Rascal to do so. Finally we will share our results and do some interpretation on these.

2 Metrics

This section will describe the metrics we measured in the given Java projects. It contains our implementation of the volume, duplication, unit size and complexity metrics listed in (Heitlager et al., 2007), together with our solution to measure this metric from the given Java projects.

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).

Definition 2.1. LoC (lines of code). The number of physical line endings ($< CR >$) in the source code.

By doing so, comments and empty lines should not be taken into account. When the LoC has been measured, the result can be rated while taking the language of the program into account. Since all given projects are Java, we used the rating system for Java (Heitlager et al., 2007).

This is implemented in Rascal by taking the *M3* object retrieved from the project and retrieving a list of all classes from it. These classes can then be read, stripped and counted. Every piece of code is stripped from spaces and tabs first and then stripped of the various types of comments in Java. When this is done, the text is split by new lines and empty lines are filtered out. This results in a list of every relevant line to count.

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

Table 1: Volume ranking scheme for Java systems.

The line count of the source code is evaluated using the SIG ranking schema in 2.1.

2.2 Unit size metric

For the unit size measure it's required to measure the LoC of all units and group these.

Definition 2.2. Unit. The minimal section of code that can executed individually.

These groups can then be used to get insights on how the code base is divided in terms of unit size (Heitlager et al., 2007). This insight can then be translated into a final rating on the unit size aspect. We used the same rating system as the unit complexity metric but adjusted the thresholds a bit to be suitable for unit size. These thresholds now start at 15 LoC or less for the best rating on unit size of a unit. This matches with the information given by Magiel in his lecture.

The LoC of each method is measured by performing the same operations as mentioned in 2.2, but then applied on all methods and constructors of the Java program. Within our implementation this is a by product of measuring the unit complexity, which we gladly use and saves us another pass through the program. Afterwards the unit sizes are paired with the percentage of the total measured LoC and poured into groups based on the set thresholds. Based on these thresholds a rating is done for the unit size metric.

2.3 Duplication metric

For the duplication metric we partition the code base in a sub list using a sliding window approach. We refer to each individual window as a block. When separate blocks contain equal content, they are considered duplicates.

Definition 2.3. Duplicate blocks The number of duplicated blocks is defined as the total number of blocks participating in duplication.

Our implementation creates blocks from six subsequent lines of code within a file. While parsing the individual lines only leading white spaces are removed when the lines are added to a block. The implemented algorithm works as follows:

Algorithm 1 Duplication detection

```
1: procedure DUPLICATES(files, blockSize)
2:     blocks ← []
3:     for file in files do
4:         i ← 0
5:         while i + size ≠ endoffile do      ▷ Iterate over the lines of the file
6:             b_i ← select(lines + i, size)      ▷ Select the number of lines
7:             blocks ← b_i
8:             i ← +1
9:     dup ← #unique(blocks)%#blocks
```

We then compare the number of duplicate blocks in the project versus the total number of blocks.

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

Table 2: Duplication ranking.

The comparison is evaluated based on the SIG scoring model as shown in table 2.4.

Because we attempt to add each block to a set we end up with a collection of unique blocks because duplicate elements are eliminated. This could be a issue performance wise because each element has to be compared to all the elements in the set before it can be considered unique and added to the set. If there was a great number of duplicates in the code base this would not be an issue because then the check could quickly terminate and return negative.

A possible workaround could be to hash the contents of a block and perform the checking on that. We tried that but had several issues with that. The hashing itself is a time intensive approach because we had to parse each char of a string and perform our magic on that. This alone made it slower than our set approach in practice.

Another issue is that approach this requires knowledge how rascal handles string comparisons. Some languages only check the first character and move on from that and/or use internal hashing methods. We currently don't know how rascal handles this. The strings we expect to find in the code are not very similar so we need to know more before we can improve our efficiency.

We did find out that it doesn't seem to make sense to use whole files for finding duplicates in case of java. In java a lot of duplications will be found by import statements alone when using total files. That's why we chose to take the class contents only.

2.4 Complexity per unit metric

For this metric we determine the cyclomatic complexity on a unit level.

Definition 2.4. Cyclomatic Complexity Representation of the number of different paths through the code. Increments by 1 for each control flow split.

The following keywords as suggested by (Landman et al., n.d.) are considered as control flow splits: if, for, do, while, case, catch, throw, foreach, &&, || and conditional.
For each file in the code base an abstract syntax tree is created. We traverse the AST (abstract syntax tree) per unit and count the number of nodes matching

one of the keywords. This count is mapped to a complexity rating. The weight of the rating of each unit is scaled by their relative size compared to the other units in the code base.

Rank	Complexity
+	>50%
o	21-50%
-	11-20%
–	1-10%

Table 3: Complexity metric ranking.

Table 2.4 shows the ranking of the complexity metric.

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 found in fig. 5 (Heitlager et al., 2007). We weighted every source code property equally and afterwards took an average of the sub attributes to get the maintainability rating.

2.6 Unit interfacing

TODO

3 results

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

3.1 Small SQL

As can be seen in the results of our analysis below, this code base is not really that maintainable. Duplicate code is really a big problem and is something that is way beyond the "+" rating threshold. Unit complexity is scored quite low, but taking a look at the actual metrics, it seems to be doable to fix this. This is due the thresholds only just being exceeded. Reducing the relative size of high and very high complexity units by only one percent will already make a huge difference in rating. Unit size though seems to be quite a big problem. There is even quite a gap in moderate sized units, so it's either small units or quite big units in this code base. Of course we must take into account as well that we did not measure the unit testing source code property, so that might change some ratings in the end.

Results

```
Volume metric:

Total lines of java code:              26246
Volume rating                          : ++

_______
Complexity metric:

Complexity groups:
Low risk percentage:          75.07470008184300
Moderate risk percentage:     8.0140926725500
High risk percentage:         11.291976988200
Very High risk percentage:    5.619230254400

Complexity rating: --
_______
Unit size metric:

Unit size groups:
Small sized percentage:             43.74080185405300
Moderate sized percentage:          13.2274896309400
Large sized percentage:             22.922891672300
Very large sized risk percentage:   20.108816839700

Unit size rating: --
_______
Duplicates metric:
```

```
Total amount of duplicate blocks:  13769
Total amount of blocks:            37493
Percentage of duplicates:          36%

Duplicates rating: --
_______
Overal scores:

Analysability: -
Changeability: --
Testability: --

Maintainability: --
_______
Bonus 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: ++

Analysis took: 14.578125 seconds
```

3.2 Hsqldb

First thing to notice is that this project is quite a lot bigger than the small SQL project. This immediately has a big impact in the time it takes to run this analysis. This is mainly caused by measuring the duplication metric. Just like the small SQL project, complexity is an issue, but not an impossible to overcome issue. Unit size is a lot worse in this project though, the relative size of very large sized units even exceeds the small sized units. Duplications is also a big problem here and exceeds the thresholds by far.

Results

```
Volume metric:

Total lines of java code:             152025
Volume rating based on lines of code: +

_______
Complexity metric:

Complexity groups:
Low risk percentage:          68.433862765741000
Moderate risk percentage:     13.10512684492200
```

```
High risk percentage:          10.60442475120000
Very High risk percentage:     7.8565856397200

Complexity rating: --

_______
Unit size metric:

Unit size groups:
Small sized percentage:                29.521640091989000
Moderate sized percentage:             15.96155646471400
Large sized percentage:                22.7740506130200
Very large sized risk percentage:      31.7427528318600

Unit size rating: --

_______
Duplicates metric:

Duplication stats:
Total amount of duplicate blocks:  100111
Total amount of blocks:            297033
Percentage of duplicates:          33%

Duplicates rating: --

_______
Overal scores:

Analysability: -
Changeability: --
Testability: --

Maintainability: --
_______
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: ++

Analysis took: 121.534045 seconds
```

References

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.