University of Amsterdam

Software Evolution

Report Series 1

Authors:
Jelle van Noord (10797483)
Katarina Lang (12336440)

November 27, 2018

Contents

1 Introduction

A goal of the course Software Evolution is to gain an overview of the quality of software systems and to pinpoint problem areas that may cause low maintainability. For that reason we implemented a software analysis in RASCAL which calculates different metrics for a Java project. We want to measure the maintainability of Java code.
The source code for the analysis can be found on GitHub[1].

2 Used metrics

Source code properties that are of interest for the project are volume, complexity per unit (unit complexity), duplication, unit size and unit interface. A unit is considered to be the smallest piece of code that can be executed and tested individually [3]. One unit in Java is a method.

Volume:
To measure the volume of a project the "simplest" way is to count the lines of code. To gain meaningful results it is important to remove comments and blank lines, since they do not influence the maintainability of code. When comparing two projects regarding their volume it is clear that the project which has more volume, also has more lines of code.

Complexity per unit:
To measure complexity per unit, we used the cyclomatic complexity. Fenton describes the cyclomatic complexity as an absolute scale measure for the attribute "number of independent paths" [2].

Duplication:
For measuring duplication we compared code blocks of at least 6 lines [3]. Only if more than 6 lines are duplicated the lines are taken into account. One reason for not taking single lines is that too many duplicates would be detected, even if they do not have a bad influence or are considered as a bad coding style. Two methods, which have the same name in different classes are one example where a duplicated line would be detected. However this does not influence the quality of the code in a bad way. Therefore blocks of six are checked for duplication.

Unit size:
For unit size we used lines of code per method. Larger units decrease the maintainability [3], therefore this number should be low.

[1] `https://github.com/jvn13/evolution`

Unit interfaces:
To measure the Unit Interfaces, the parameters of each method are counted. Methods should have a low amount of parameters, because that makes them easier to understand, easier to test and foremost easier to reuse.

3 Computation of the metrics

3.1 Concrete values

Lines of code (for volume and unit size):
One counted line is similar to a line in a java file. To get those lines it is useful to use the rascal function *readFileLines(loc location)*. It returns a list of all lines in a java project. When counting the lines of code it is important to remove lines which do not influence the code quality. That are mainly comments and empty lines.

Since there are many ways to comment in a Java file, each possibility should be taken into account. Each line of the Java file is tested for comments. If there is a comment, it will be removed. In the end a list of lines of code of each method is calculated. The methods are provided by the Rascal m3. The size of the list of lines is returned for each method to get the unit size. To get the lines of code for a whole project the lines of code are generated per file and added to a list. In the end the size of this list is returned.

Cyclomatic complexity:
To compute the cyclomatic complexity the Rascal AST is used. It is a symbolic representation for abstract syntax trees of programming languages [1]. The interesting part is the Declaration, because this one specifies methods, interfaces and constructors. To calculate the cyclomatic complexity the methods are of special interest.

For each method the nodes are visited. Every time an if, for, foreach, while, catch, case or infixes like "&&" appear, the cyclomatic complexity is increased by 1. The cyclomatic complexity is computed for each method of the project and is added to a list. This list represents the cyclomatic complexity of each method of the Java project.

Duplication of blocks of six:
For the duplication, the source code lines (non-empty and non-comment) of each file are appended to a *list[str]*. Blocks of six lines are created by looping over the *list[str]* of lines and then appending the successive five lines into one large String. This String is used as the key in a *map[str,list[list[int]]]*.

For each block the algorithm checks if the key already exists. If no key is found, it means that it is the first time the algorithm encounters the block. In this case a map entry is added, where the list of indices of the lines in the block is the value of the entry.

If a key is found it means that the algorithm encountered the block before and that it found a redundant block. In this case the list of indices of the redundant block lines is added to the value of the existing map entry.

The result of the algorithm is a map which has the blocks of six lines as keys and the values of these keys are lists of each occurrence. Each list consists of a list of line indices of the block. Keys which have only one value, occurre only once in the project and are removed from the map. The result of this is a map of the duplicated blocks.

Afterwards the algorithm unravels all the values of the map and appends them to a set. Since a set can not contain duplications, each line can be counted only once as duplicated, even if it appears more than once in the map. These duplicates are unavoidable since some duplication blocks have some overlap. Duplicated lines can occur in blocks larger than six lines, but the algorithm only selects six lines.
The size of the set is returned and set in contrast to the overall LOC.

Unit interface:
The metric for unit interface is calculated by taking the first source code line (SLOC) of each method, since this is the declaration of the method. In that line the program analyses the part between parenthesis, because this part contains the parameters. It will split the found string on commas and count each non-empty part as a parameter. The metric for unit interface is calculated at the same time as the one for unit size for performance reasons. The algorithm already goes over each line when calculating the unit size.

3.2 Risk Profiles

Unit complexity:
For the risk profile the following schema was used according to an article by Heitlager, Kuipers and Visser [3]. It is usable for every language and is very clear.

If a unit has a complexity that is less than 10 it is simple and does not contain much risk. If it is between 11 and 20 it is more complex and has a moderate risk. Methods with a cyclomatic complexity between 21 and 50 are evaluated as complex and have a high risk. For complexities higher than 50 the unit is considered as untestable and has a very high risk. The exact profile can be found in table 1.

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 1: Risk Evaluation: Unit complexity

Unit size:
For this risk profile the profiles of the book Building Maintainable Software [4] are used. The authors state that people learn to code efficiently and therefore set the scores relatively low compared to previous profiles.

The final profile considers a unit size of 1-15 as simple, a unit size of 16-30 as more complex, a unit size of 31-60 as complex and attached to a high risk and a unit size bigger than 60 as untestable and incidental with a very high risk. The exact profile can be found in table 2.

Unit size	Risk Evaluation
1-15	simple, without much risk
16-30	more complex, moderate risk
31-60	complex, high risk
>60	untestable, very high risk

Table 2: Risk Evaluation: Unit size

Unit Interfaces:
For this risk profile the profiles of the book Building Maintainable Software [4, chapter 5] are used. The authors argue that less parameters will make the units more maintainable and more reusable. Units with a lot of parameters should be replaced by objects.

The final profile considers a unit interface of 1-2 as simple, a unit interface of 3-4 as more complex, a unit interface of 5-6 as complex and every unit interface greater than 6 is considered untestable and of high risk. An overview of the boundaries is given in table 3.

Unit interfaces	Risk Evaluation
1-2	simple, without much risk
3-4	more complex, moderate risk
5-6	complex, high risk
> 6	untestable, very high risk

Table 3: Risk Evaluation: Unit interfaces

3.3 Scores / Rank:

To get a general rating and an overview of all the metrics and risks, scores are used. This scores are calculated based on the SIG model [3]. There are five possible ranks (- -, -, o, +, ++). Five different outcomes are useful, since it is still possible to get an overview, while it is not too complex. If e.g. only good, (neutral,) and bad are used, it is too simple to differentiate between different projects. However the neutral rank is useful if no statement / decision can be made.

Volume:
The volume rating is based on the SIG model [3]. If a project contains more lines it is more likely that more risks are connected with it. The more lines there are the more tests are needed and the more dependencies are present. An overall KLOC of less than 66 is ranked as very good. KLOC up to 246 is ranked as good, while a number between 246 and 665 is rated neutral. If a project has more than 665 KLOC, but less than 1310 it is bad. Otherwise it is very bad. The exact scoring profile can be found in table 4.

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

Table 4: Rating: Volume

Unit complexity, size and interfaces:
The percentage for the scores of unit complexity, unit size and unit interfaces are based on the SIG model [3]. If a project has more units, that are rated with a very high risk the overall rank result should be lower than the one for projects with less critical units.

To gain the risk profile the lines per risk class are counted and set in contrast to the overall loc. It is important that the overall loc is used, not only the lines of all methods. Since the complexity should be calculated for the whole project it makes sense to take every line into account (except from comments and empty lines). The profile for the unit complexity, unit size and unit interfaces can be seen in Table 5. To get a very good score a project should have less than 25 % of its lines in units with moderate risks and 0 % in units with high risks and very high risks.

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

Table 5: Rating: Unit complexity, unit size and unit interfaces

Duplication
The ranks for duplication are based on the SIG model [3]. The more duplication a project has, the lower is its rank. The use of percent values has an important reason. It makes a difference if there is one duplicated block of six lines in a project with 12 lines or in one with 1200 lines. If analyzing the duplicated lines the total number of lines in the project is taken into account.

If a project has 0 - 3 % duplication, it gets a very good rank. Up to 5 % it is good, up to 10 % neutral, up to 20 % negative and everything else is very negative.The exact scoring profile can be found in table 6.

rank	duplication
++	0-3%
+	3-5%
o	5-10%
-	10-20%
- -	20-100%

Table 6: Rating: Duplication

4 How well do these metrics indicate what we really want to know about these systems and how can we judge that?

If we want to know something about the volume of a project, we use a metric such as lines of code. The result is a number. The task now is to understand the metric. Which number indicates a high volume and when does a project have a small volume?

When using different metrics the result can be different. Moreover you need to know how to interpret those results. The metrics are always approximations and must not be accurate. Most of the time the metric is useful to measure a specific aspect. Special cases are not taken into account. One example is for the cyclomatic complexity if the code contains a statement like *if(true)*.

5 How can we improve any of the above?

Improving volume:
Different coding styles lead to a different number of lines of code for the same content in a java project. One option to improve the metric and to make it independent from coding styles is to transform the whole code into a general pattern.

One example are switch-cases. It is possible to write the return statement in the same line as the case. An other option is to use two lines. There are more of those examples, which could influence the number of lines. It is important to get a metric which does not dependent on the coding style of a programmer if you want to compare projects based on the lines of code.

Another example are closing brackets. They can be in an additional line or be at the end of the previous line.

If there is a comment inside a quote it will be removed in our implementation. One approach to improve the duplication metric is to use pattern matching to detect whether a comment should be removed or not.

Improving duplication:
The duplication metrics can be improved by making it name independent. If there are two methods of six lines that differ only in their method name they would not be detected as duplicates. Another example is two methods which only differ in the name of their variables but still do the same thing. It is a duplication in both cases, but not detected in our solution.

6 Scoring / Rating

One aspect of external and internal quality of software is maintainability. Maintainability is split up into analysability, changeability, stability and testability [3].

The implemented solution calculates scores for the following maintainability aspects based on the SIG model:

- Analysability
- Changeability
- Testability
- Reusability

Afterwards those scores are combined to calculate a score for the maintainability of the Java project which is evaluated. To calculate Analysability, Changeability and Testability the metrics for volume, unit complexity, unit size and duplication can be used. Reusability additionally takes unit interface into account. Not every metric from above has a direct influence on the maintainability aspects. Therefore it is important to indetify which metrics affect which maintainability aspect. The SIG model [3] is used to determine if a measure influences a maintainability aspect.

	volume	unit complexity	duplication	unit size	unit interfaces
analysability	x		x	x	
changeability		x	x		
testability		x		x	
reusability				x	x

Table 7: Mapping system characteristics (left) onto source code properties (right) [3]

The following formulas were used to calculate the aspects:

Analysability:
Analysability = 0.4 * volume + 0.2 * duplicates + 0.4 * unit size

The larger the system and its units gets the more difficult it becomes to identify parts which need to be changed. Duplication is also important since a change in the "original" code part could also result in a need to change the duplicated part. It is difficult to find that part of the code.

Testability:
Testability = 0.7 * complexity + 0.3 * unit size

Since the cyclomatic complexity measures independent paths through a system it gives an indication on how many tests are needed at least. The higher the CC, the more tests are needed and therefore the harder the project is to test. Especially after a modification is made, many tests need to be run again to check the code.

Changeability:
Changeability = 0.5 * complexity + 0.5 * duplication

Both the aspects are important when calculating the changeability. If a unit with a high complexity should be changed, it is more difficult to think about the effects than changing a unit with a low complexity. Also the duplication has an influence on the changeability since all duplicates are likely need to be also changed.

Reusability:
Reusability = 0.5 * unit size + 0.5 * unit interface

Small units will have a positive effect on reuseability, when more functionality is added to the system [4]. Since the small units contain less functionality and are therefore general, they can be used for other purposes.

If methods contain a large list of input parameters, they are very specific and limit the possibilities to reuse them. By keeping the number of input parameters low, the methods stays maintainable and can be reused in different cases.

Maintainability:
Maintainability = 0.25 * Analyseability + 0.25 * Testability + 0.25 * Changeability + 0.25 * Reuseability

As stated in the introduction all the metrics have an influence on software maintainability and should therefore be considered when calculating software maintainability.

7 Result

The program is used for two open source projects: smallsql and hsql. [2]

7.1 smallsql

Table 8 shows the metrics and their values and ratings computed out of all the Java files in the smallsql project.

Metric	Absolute value	Rating
Volume	24010	++
Unit Size	8.782 (avg)	–
Unit Complexity	2.754 (avg)	-
Duplicates	2692 (11.21%)	-
Redundants	1602 (6.67%)	n/a
Unit Interfaces	0.773 (avg)	+

Table 8: Values and ratings for metrics of the smallsql project

[2] `https://homepages.cwi.nl/~jurgenv/teaching/evolution1314/assignment1.zip`

Table 9 displays the relative risks percentages of the Unit Complexity, Unit Size and Unit Interfaces for the smallsql project. The Volume, Duplicates and Redundants do not require a risk profile, because their scores / ratings are based on the absolute values.

	Risk levels			
Metric	Low	Moderate	High	Very high
Unit Complexity	62%	7%	10%	5%
Unit Size	40%	19%	14%	14%
Unit Interfaces	79%	8%	1%	0%

Table 9: Risks of the metrics (left) of the smallsql project

Table 10 displays the ratings of the smallsql project for the chosen characteristics. An explanation of the calculation of these can be found in section 6.

Characteristic	Rating
Analysability	o
Changebility	-
Testability	-
Reusability	-
Maintainability	-

Table 10: Ratings for the characteristics of the smallsql project

7.2 hsql

Table 11 shows the metrics and their values and ratings computed out of all the Java files in the hsql project.

Metric	Absolute value	Rating
Volume	171470	+
Unit Size	13.951 (avg)	- -
Unit Complexity	3.757 (avg)	- -
Duplicates	34960 (20.39%)	-
Redundants	20869 (12.17%)	n/a
Unit Interfaces	0.878 (avg)	++

Table 11: Values and ratings for metrics of the hsql project

Table 12 displays the relative risks percentages of the Unit Complexity, Unit Size and Unit Interfaces for the hsql project.

	Risk levels			
Metric	Low	Moderate	High	Very high
Unit Complexity	53%	12%	10%	8%
Unit Size	27%	17%	16%	28%
Unit Interfaces	82%	6%	0%	0%

Table 12: Risks of the metrics (left) of the hsql project

Table 13 displays the ratings of the hsql project for the chosen characteristics. An explanation of the calculation of these can be found in section 6.

Characteristic	Rating
Analysability	-
Changebility	-
Testability	- -
Reusability	o
Maintainability	-

Table 13: Ratings for the characteristics of the hsql project

For completeness screen shots of the results of the program after analyzing both of the projects are appended in the Appendix.

References

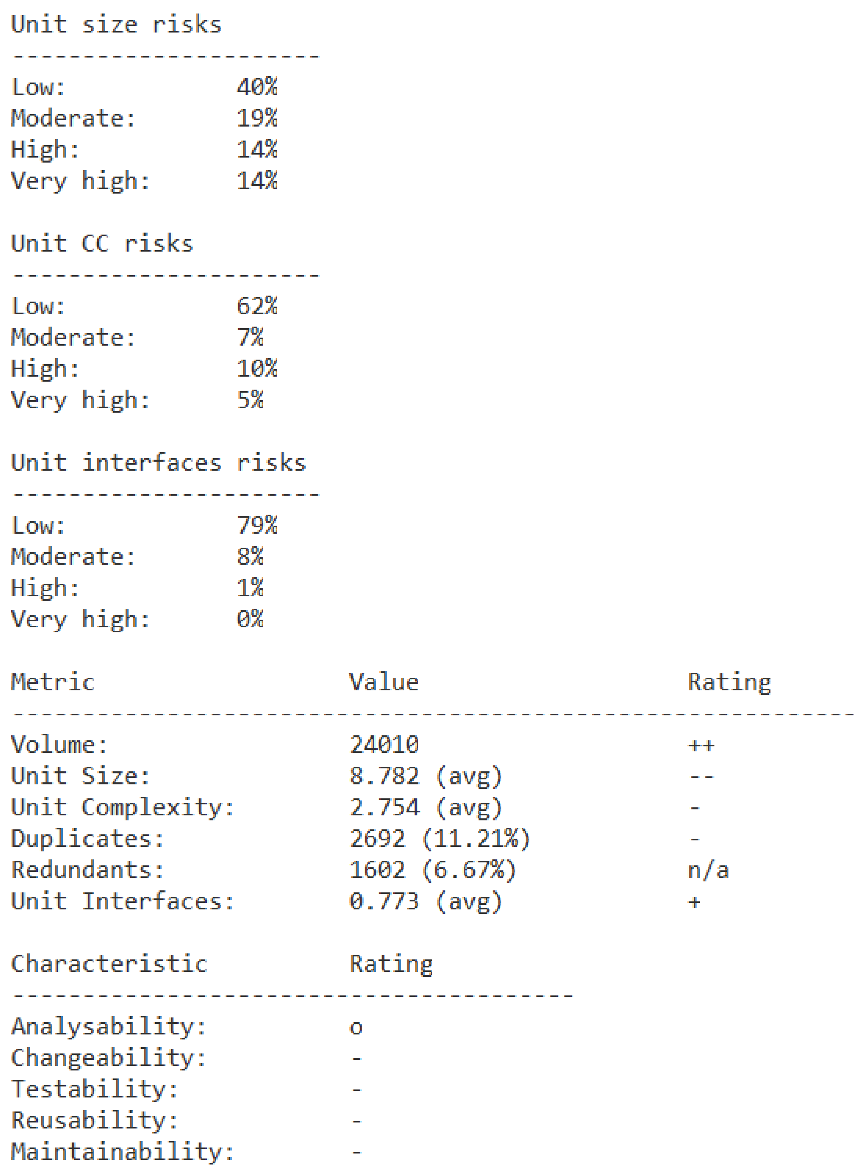
[1] `http://tutor.rascal-mpl.org/Rascal/Libraries/Prelude/IO/readFile/readFile.html#/Rascal/Libraries/analysis/m3/AST/AST.html`.

[2] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. Softw. Eng.*, 20(3):199–206, March 1994.

[3] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 30–39, Sept 2007.

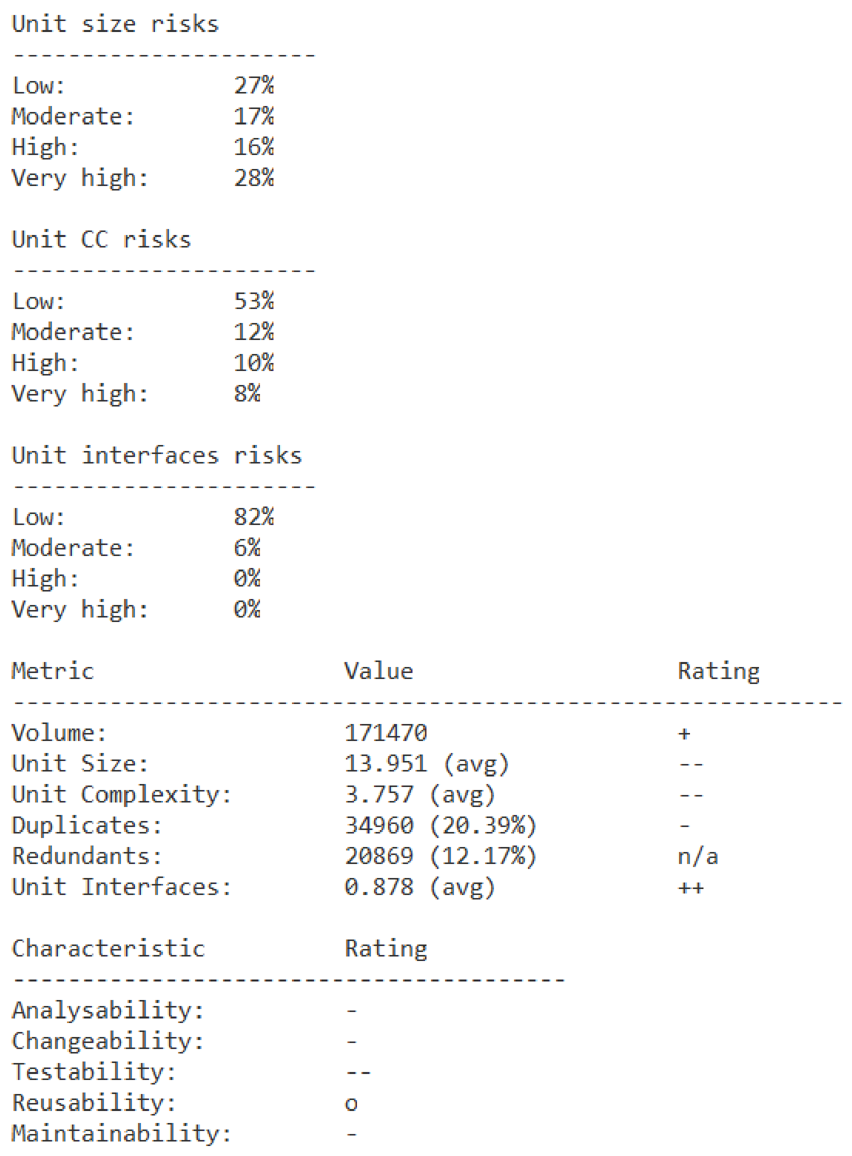
[4] Joost Visser, Sylvan Rigal, Rob van der Leek, Pascal van Eck, and Gijs Wijnholds. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code.* O'Reilly Media, Inc., 1st edition, 2016.

Appendix



```
Unit size risks
----------------------
Low:            40%
Moderate:       19%
High:           14%
Very high:      14%

Unit CC risks
----------------------
Low:            62%
Moderate:       7%
High:           10%
Very high:      5%

Unit interfaces risks
----------------------
Low:            79%
Moderate:       8%
High:           1%
Very high:      0%

Metric                  Value                   Rating
------------------------------------------------------------
Volume:                 24010                   ++
Unit Size:              8.782 (avg)             --
Unit Complexity:        2.754 (avg)             -
Duplicates:             2692 (11.21%)           -
Redundants:             1602 (6.67%)            n/a
Unit Interfaces:        0.773 (avg)             +

Characteristic          Rating
----------------------------------------
Analysability:          o
Changeability:          -
Testability:            -
Reusability:            -
Maintainability:        -
```

Figure 1: Program output after analyzing the smallsql project



```
Unit size risks
----------------------
Low:            27%
Moderate:       17%
High:           16%
Very high:      28%

Unit CC risks
----------------------
Low:            53%
Moderate:       12%
High:           10%
Very high:      8%

Unit interfaces risks
----------------------
Low:            82%
Moderate:       6%
High:           0%
Very high:      0%

Metric                  Value                   Rating
------------------------------------------------------------
Volume:                 171470                  +
Unit Size:              13.951 (avg)            --
Unit Complexity:        3.757 (avg)             --
Duplicates:             34960 (20.39%)          -
Redundants:             20869 (12.17%)          n/a
Unit Interfaces:        0.878 (avg)             ++

Characteristic          Rating
----------------------------------------
Analysability:          -
Changeability:          -
Testability:            --
Reusability:            o
Maintainability:        -
```

Figure 2: Program output after analyzing the hsql project