Software Evolution

Series #1

Jordan Maduro
10807411
jordanmaduro@gmail.com
Software Engineering

Cas van der Weegen
10017577
cas.vanderweegen@student.uva.nl
Software Engineering

November 23, 2017

UNIVERSITY OF AMSTERDAM

Contents

1 Summary

The Series #1 assignment consists of implementing a set of metrics that can be used to collect empirical data from a software project (e.g. its source-code). The overall goal of the assignment is twofold:

1. To get acquainted with RASCAL, an experimental meta-programming language that can be used for static-code analysis, program transformation, and the implementation of domain-specific languages [1].

2. To get familiar with code-quality, the means of measuring code quality, and the *SIG Maintainability Model* [2].

For two software projects that were provided as part of the assignment several metrics were collected that are part of the SIG Maintainability Model, which in turn is based on the *ISO/IEC 9126 standard* [3]. The two projects provided are:

1. SmallSQL - a 100% pure Java database for Java desktop applications [4].

2. Hyper SQL Database - HSQLDB - 100% Java Database. [5].

The projects differ in greatly in size which makes their combination useful for testing our analysis implementation (SmallSQL is relatively small, as the name suggests, HSQLDB is quite a bit larger).

2 Metrics

2.1 Volume

The overall volume of the source code influences the analysability of the system since, in general, a larger system required a larger effort to maintain [2]. Our implementation calculates the following volume metrics of a project:

1. Total lines - the total number of lines in the project

2. Source lines - the total number of actual source code (functional lines)

3. Comment lines - lines that are either single-line comments or commented out code

4. Comments - all comments (including multi-line comments)

Our approach is as follows: First, we create an M3 tree using the *createM3FromEclipseProject* function in rascal, the function *files* is subsequently used to pass a set of locations to our volume function. Secondly, we filter out all comment lines and empty lines using regexes. Lastly, based on the results obtained from the previous step we calculate all the different values we need to say something useful about the source-code.

The output of our volume function is a *map[str, int]* that contains the metrics described above. Our visualization counterpart of the volume function converts this into a scale that can be used to calculate the SIG maintainability score.

2.2 Complexity per Unit

The complexity of source code units influences the system's changeability and its testability [2] since the more complex a unit is, the more difficult it is to understand by and a programmer (which means changing the code, or writing tests for it becomes more complex).

Complexity (cyclomatic) can be calculated by determining the size of the control flow graph, we took the following approach to do so: First, we generated asts of the project using the *createAstsFromEclipseProject* function in RASCAL, subsequently the *methods* function is called which returns a set of *Declarations* (set[Declaration]) and the result is passed to our complexity function. Second we calculate the complexity by calculating the number of statements that cause the control flow graph to increase in size (i.e. if, while, case, do, for, foreach, conditional, && (AND), and || (OR)). Additionally the volume of the method (the number of source lines) is calculated using our volume function.

Both values are returned are an *lrel[int, int]* such that we can later calculated the relative complexity per method, which will be handled by the counterpart class in the *visualize* package.

2.3 Duplication

The duplication of source code influences analysability and changeability of the system [2]. Duplications lead to a system being larger than it needs to be. The SIG model of maintainability defines duplication as a block of 6 or more lines that occur more than once.

We used a sliding window approach to finding duplicate blocks. For each file, we normalize the source and create chunks of 6 lines. Rather than using a hash function we use the concatenated source as the hash. To be able to identify each source line we use the line- and file-index as an identifier. Depending on the chosen window size (W), and source code lines (S), we could get a massive amount of blocks (N) which is visualized in 1.

$$N = 1 + (S - W) \tag{1}$$

Some blocks overlap and contain the same lines, and others contain no match.

To improve performance, we reduce the search space by only looking for blocks that occur more than once. We do this using the distribution function. We use

rangeX to remove all blocks of equal length to the window size.

We build a list of all source lines within the remaining blocks and remove all duplicates using the dup function. The source line identifier allows us to avoid adding the same line more than once to the duplicate count. The size of the remaining list gives us the number of duplicate lines.

```
Alternative Implementation
  SmallSQL:
    Duplicate lines        : 2706
    Duplication            : 11.22%
    Time Spent             : 1s

  HSQLDB:
    Duplicate lines        : 28900
    Duplication            : 16.67%
    Time Spent             : 21s

Initial Implementation
  SmallSQL:
    Duplicate lines        : 2560
    Duplication            : 10.62%
    Time Spent             : 7s

  HSQLDB:
    Duplicate lines        : 28317
    Duplication            : 16.33%
    Time Spent             : 224s
```

We believe that the statistical error that occurs by using the alternative implementation is acceptable for most cases, especially seeing that the percentages of classification are tweaked on a regular basis [6][7][8]. However, for the completeness of this assignment we chose to use the more accurate (but slower) implementation to calculate our final results.

2.4 Unit Size

The size of units influences their analysability and testability and therefore of the system as a whole [2] since the larger a unit it, the more difficult it is to understand its function or the correct way to test it. Additionally, it can be a sign that a method can be divided into smaller units.

Our implementation for calculating the unit size utilizes the functions written for the volume metric but instead of calculating the size on a per-file base (as the volume metric does) it calculates the size per method. For that reason

the approach is nearly the same with the only difference being that instead of passing the files (using the *files* function) we pass the methods of a project (using the *methods* function). The result of the unitsize function is a list of integers, which is to be parsed by its visualize counterpart.

2.5 Unit Testing

The unit testing metric influences testability, stability, and analysability[2]. We decided not to use Clover or other tools to dynamically measure the test coverage of the unit tests. We did however implement a Test Quality metric. Our approach to this metric is to identify Unit Tests classes. For this assignment, we chose to follow the JUnit conventions.

First, we visit the AST to find all classes with names starting with "Test". Second, each class is visited to extract all methods which start with "Test". Finally, we calculate the quality of the unit test by counting the percentage of unit test methods which contains at least one assert. The result is ranked based on the test coverage table[2].

2.6 Additional Metrics

2.6.1 Number of Methods

The Number of Methods (NOM) influences the analysability of a class. We implemented this metric by using the M3 to retrieve all the classes. Then extract all methods per class.

2.6.2 Weighted Methods per Class

The Weighted Method Per Class (WMC) gives information about the analysability of a class[9]. We implemented the WMC using the cyclomatic complexity to weigh each method. First, we extract the AST. Second, we calculate the cyclomatic complexity using UnitComplixty. Finally, return the sum of the complexities per class.

2.6.3 Duplication (alternative)

Based on our initial results we decided to profile our code to figure our which function could be further optimized. Based on the profiling we discovered that our duplication function (the function that calculates the number of duplicate source lines) was utilizing relatively much time to complete, specifically with the creation of blocks of code (our window of 6 lines).

Our solution was to decrease the number of loops, recursive calls, and list-operations our code was performing to reduce the overall computation time. This lead us to implement a function that create blocks and subsequently a hash of this block which is then to be used in further computations. No having

to re-calculate the block several times changed the complexity of the duplication function from O^2 to O.

The downside of this implementation is that it is less accuracy since the calculations performed with the hashes are less accurate then actually visiting each line, this lead us to utilize our initial implementation and instead dedicate this section to our alternative code instead of using it in the final code.

3 Visualization

To visualize our metrics according to the SIG maintainability model we chose to create a separate package in our framework, with the sole responsibility of visualizing the metrics collected and calculating the corresponding SIG Maintainability aspects.

The *visualize* package of our framework consists of two parts, the *aspects* and the *metrics* subpackages. The metrics package is responsible of classifying the actual date obtained from the metrics (package) classes according to [2][6][7][8], and are assigned a class from -2 to 2 [1] (zero to five stars) based on the following metrics:

- Volume
  Based on a predefined table a class is assigned based on the number of source-code lines in the project. The reference table is defined in [2] as shown in table 1.

Rank	Lines of Code
++	0 - 66000
+	66000 - 246000
o	246000 - 665000
-	655000 - 1310000
- -	>1310000%

Table 1: Volume Classification

[1] a scale from -2 to 2 was chosen, instead of the regular 0 to 5, to make the omission of metrics easier such that it does not skew the results

- Complexity per Unit
  Each unit in the source code (method) is assigned a risk class based on the complexity calculated in the metrics package (shown in table 2). The number of source lines belonging to the class is then added to the calculated risk class such that a representative overview of lines-per-risk-class can be calculated. The result of the above classification is then mapped onto a risk-table provided by [2] Each unit is classified, and together with

Complexity	Evaluation
1-10	Little to no risk
11-20	Moderate risk
21-50	High risk
>50	Very high risk

Table 2: Complexity Risk Classification

  the size of the unit (which is obtained through the volume metric) a classification to maximum relative LOC is done and awarded a rank, based on table 3.

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

Table 3: Complexity Rank Classification

- Duplication
  To classify the duplication metric, we took the percentage of duplicate source lines and assigned a rank based on table 4[2].

Rank	Percentage
++	0-3%
+	3-5%
o	5-10%
-	10-20%
- -	20-100%

Table 4: Duplication Rank Classification

- Unit Size
  To classify the unit size for each unit source-code lines are calculated (using the volume function). Each unit size is then classified using table 5 and assigned a risk profile (which is defined by the SIGTUViT [8]). Each

Complexity	Evaluation
1-30	Little to no risk
30-44	Moderate risk
44-74	High risk
>74	Very high risk

Table 5: Unit Size Classification

  of the risk profiles have a threshold for which a penalty is given to the class (i.e. the class is reduced by one), the thresh are defined in table 6. Additionally, an additional point is deduced if there are *any* high risk

Risk	Penalty Threshold
moderate risk	$> 44.4\%$
high risk	$> 21.4\%$
very high risk	$> 7\%$

Table 6: Unit Size Rank Classification

  units. Using the following method we can calculate a class ranging from -2 to 2.

- Unit Testing
  To classify the unittesting metric we decided to calculate the code coverage of the unittests for a project. Based on the code coverage of the projects a class is assigned (again from the range -2 to 2), based on table 7 which is defined in [2].

Rank	Percentage
++	95-100%
+	80-95%
o	60-80%
-	20-60%
- -	0-20%

Table 7: Unit Testing Rank Classification

For the purpose of this assignment we implemented the following 4 aspects, each of which is part of the *SIG Maintainability Model*.

- Analysability
  composed of the average of volume, duplication, unit size, and unit testing.
- Changeability
  composed of the average of unit complexity and duplication.
- Stability
  based on the unit testing class.
- Testability
  composed of the average of unit complexity, unit size, and unit testing.

From each of the metric classes the average is taken and rounded *down*, this value is the score for this specific aspect. Subsequently an average is taken from all the aspect scores, which results in the SIG score that is awarded to the project.

4 Results

The following results were obtained using both the *small* project [4] and the *large* project [5] using a machine of the following specifications:

```
Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz, 16GB RAM, 512GB SSD Storage
```

4.1 SmallSQL

```
NOM
  Classes          : 192
  Methods          : 2401
  Max              : 167
  Average          : 12

WMC
  Classes          : 192
  Max              : 555
  Average          : 31

Volume
  Class            : 2
  Rank             : ++
  Total lines      : 38423
  Source lines     : 24107

Unit Complexity
  Class            : -2
  Rank             : --
  Percentages
    No risk        : 73.47%
    Low risk       : 8.47%
    Medium risk    : 12.08%
    High risk      : 5.99%

Unit Size
  Class            : 0
  Rank             : o
  Percentages
    No risk        : 62.61%
    Low risk       : 11.39%
    Medium risk    : 12.71%
    High risk      : 13.29%

Unit Test quality
  Class            : 1
  Rank             : +
  Avg. Percentage: 84.74

Duplication
  Class            : -1
  Rank             : -
  Percentage       : 10.62%
  Duplicates       : 2560

Analysability
  Class            : 0
  Rank             : o
  SIG Score        : ***

Changeability
  Class            : -1
  Rank             : -
  SIG Score        : **

Stability
  Class            : 1
  Rank             : +
  SIG Score        : ****

Testability
  Class            : 0
  Rank             : o
  SIG Score        : ***

SIG Grade          : ***

Time taken 34 seconds
```

4.2 HSQLDB

NOM
- Classes : 623
- Methods : 11067
- Max : 223
- Average : 17

WMC
- Classes : 623
- Max : 1312
- Average : 59

Volume
- Class : 1
- Rank : +
- Total lines : 299727
- Source lines : 173369

Unit Complexity
- Class : −2
- Rank : −−
- Percentages
  - No risk : 62.94%
  - Low risk : 15.16%
  - Medium risk : 12.07%
  - High risk : 9.82%

Unit Size
- Class : 0
- Rank : o
- Percentages
  - No risk : 35.89%
  - Low risk : 14.08%
  - Medium risk : 15.81%
  - High risk : 34.21%

Unit Test quality
- Class : −1
- Rank : −
- Avg. Percentage: 47.13

Duplication
- Class : −1
- Rank : −
- Percentage : 16.33%
- Duplicates : 28317

Analysability
- Class : 0
- Rank : o
- SIG Score : ***

Changeability
- Class : −1
- Rank : −
- SIG Score : **

Stability
- Class : −1
- Rank : −
- SIG Score : **

Testability
- Class : 0
- Rank : o
- SIG Score : ***

SIG Grade : ***

Time taken 486 seconds

Bibliography

[1] Paul Klint, Tijs van der Storm, and Jurgen Vinju. *RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation.* SCAM 09, pages 168177, Washington, DC, USA, 2009. IEEE Computer Society.

[2] I. Heitlager, T. Kuipers, and J. Visser. *A Practical Model for Measuring Maintainability.* Quality of Information and Communications Technology, 2007.

[3] ISO/IEC 9126-1:2001 (2001). Software engineering – Product quality – Part 1: Quality model. Standard, International Organization for Standardization, Geneva, CH.

[4] Volker Berlin (2005). SmallSQL (version 0.21) [Computer software]. Berlin, Germany. http://www.smallsql.de.

[5] The HSQL Development Group (2001). Hyper SQL Database (version 2.3.1) [Computer software]. London, United Kingdom. http://www.hsqldb.org.

[6] Software Improvement Group (SIG) and TV Informationstechnik GmbH (TViT). (2009). SIG/TViT evaluation criteria – Trusted Product Maintainability, version 7.1.

[7] Software Improvement Group (SIG) and TV Informationstechnik GmbH (TViT). (2009). SIG/TViT evaluation criteria – Trusted Product Maintainability, version 8.0.

[8] Software Improvement Group (SIG) and TV Informationstechnik GmbH (TViT). (2009). SIG/TViT evaluation criteria – Trusted Product Maintainability, version 9.0.

[9] S. R. Chidamber and C. F. Kemerer. 1994. A Metrics Suite for Object Oriented Design. IEEE Trans. Softw. Eng. 20, 6 (June 1994), 476-493. DOI=http://dx.doi.org/10.1109/32.295895