Series 1

This documents contains our notes and answers to the questions about software metrics (practical lab Series 1).

Authors

- Cornelius Ries
- Piotr Kosytorz

Decisions and motivations

Which metrics are used?

The following metrics are calculated:

- Volume,
- Unit Size,
- Unit Complexity,
- Duplication,
- Test Quality,
- Unit Interfacing.

SIG model metrics:

- Maintainability (overall),
- Analysability,
- Changeability,
- Testability
- Stability.

How are these metrics computed?

We used the following strategies to count specific metrics:

Volume

The basic measure for volume (according to [1]) is the number of lines of code in the project. In our soultion we count all lines of code in the whole project of all `.java` files (including tests). To reach the best results we purify the source code files before counting the number of lines per file by: * Trimming all lines of code (removing white spaces from the beginning and the end of line) * After trimming - removing all empty lines in given file * Removing single-line comment (where the line starts with `//`) * Removing multi-line comments - `/* ... */`(in all variants such as a comment beginning in one line just after code, etc.) **We do count curly braces** as lines of code.

Volume rating

As given in [1], we use the following table to as conversion basis to obtain the SIG volume score:

rank	MY	Java KLOC
++	$0-8$	0-66
+	$8-30$	66-246
o	$30-80$	246-665
-	$80-160$	655-1,310
–	> 160	$> 1,310$

Unit Size

Simillary to volume, Uint size is a count of lines of code per unit. We use Rascal's AST parser and retrieve units it. We purify each unit in simmilar way as described above (see: Volume) and count number of lines per unit.

For benchmarking the unit size (possibly simmilar to SIG standards) we have used the following tresholds taken from [3]:

CC	Risk evaluation
< 30	simple, without much risk
30-44	more complex, moderate risk
44-74	complex, high risk
> 74	untestable, very high risk

Maximum relative LOC:

rank	moderate	high	very high
++	19.5%	10.9%	3.9%
+	26.0%	15.5%	6.5%
o	34.1%	22.2%	11.0%
-	45.9%	31.4%	18.1%
–	-	-	-

Unit Complexity

The default code complexity per unit is defined to be 1. A unit in this case are methods and constructors. Based on information provided in [1] and [2], we decided to count the following statements as an increment of code complexity per unit: * case * catch * do * if * conditional * for * foreach * while * && * ||

The original rascal code responsible for counting of cyclomatic complexity:

```
int cc = 1;
visit(implementation) {
    case /\case(_)          : cc += 1;
    case /\catch(_,_)       : cc += 1;
    case /\do(_,_)          : cc += 1;
    case /\if(_,_)          : cc += 1;
    case /\if(_,_,_)        : cc += 1;
    case /\conditional(_, _, _) : cc += 1;
    case /\for(_,_,_,_)     : cc += 1;
    case /\for(_,_,_)       : cc += 1;
    case /\foreach(_,_,_)       : cc += 1;
    case /\while(_,_)       : cc += 1;
    case \infix(_, /^\|\||&&$/, _)  : cc += 1;
}
```

According to [1] we perform the following operations to optain the SIG score for CC:

First: Evaluate cc risks per unit based on thresholds from the following table:

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

Finally: Score units per number of units falling int the following tresholds:

Maximum relative LOC:

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

Test Quality

For test quality we count all the assert statements in test classes in the code [1]. After counting the assert statements we calculate the percentage based on the total number of units in the system.

For benchmarking the test quality we have used the following tresholds:

rank	percentage
++	95-100%
+	80-95%
o	60-80%
-	20-60%
–	0-20%

Unit Interfacing

For information on how to compute this metric we have looked for different papers and found [3]. For unit interfacing we count all the parameters for all methods in the system using the AST. After gathering the information we calculate a risk profile based on the following scheme:

We seperate the units based on the information below and calculate a percentage against the whole number of units.

number of parameters	Risk evaluation
< 2	without much risk
$== 2$	moderate risk
$== 3$	high risk
> 4	very high risk

After that we calculate the score based on the following thresholds:

rank	moderate	high	very high
++	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%
–	-	-	-

Duplication

We've came up with three different ways of counting duplicaed lines. Different methods can lead to very different results, which shows that counting duplicated lines based only on the textual representation of tested programs is prone to errors and should be taken with much reserve.

Method 1: Comparing duplicated blocks

The first and the easiest way that we came up with was to extract code blocks from the AST and then, after code purification per block, we were simply comparing the blocks whether they contain each other - if yes, then such a block wyould be trated as a duplicated one.

Method 2: Line per line text searching (top-to-bottom)

This method has delivered the most code duplicated blocks, but is extremaly slow, as it requires to compare most of lines with each other.

The algorithm:

1. Purify the code by removing comments and empty lines and trimming every line.
2. Store all files (as lists of lines) in one list of lines (combo).
3. Iterate from the top of the list and compare each line with all lines that are placed beneath it.
4. When the lines are idential then compare the consecutive lines of each parts (the one that you iterating over and the one that you are comaparing) - in other words - expand the comparision window.
5. Each window that is longer than 5 lines is marked as a duplicate, and all lines that have been deteced as duplicates are marked, so that they won't be used as comparision source for further comparision.

Method 3: 6-lines duplication cadidates*

We eventually decided to take another approach that counts code duplicates in decent time.

The algorithm is presented below:

1. Purify the code by removing comments and empty lines and trimming every line.
2. For all files create a list of blocks of 6 consecutive lines and save the line numbers and file locations where they start.
3. Merge all those files into one big list
4. Create a list of clone candidates woth the following method: `cloneCandidates = distribution(blob.content - dup(blob.content));`, which means: show the distribution of blocks that are duplicated. The number of occurences of certain block in this operation will be equal to numbers of copies of a certain block.
5. In extracted list of blocks, merge all blocks that start on consecutive lines of the same file (to achieve the biggest possible chunks of duplicated code).
6. Finally we sum up the number of lines of the extracted chunks.

Maintanability

To calculate the maintainability scores we compute the avarage of the relevant scores. Those are:

- Maintainability : Volume, Unit-Cyclomatic-Complexity, Unit-Size, Duplication, Test-Quality
- Analysability : Volume, Unit-Size, Duplication, Test-Quality
- Changeability : Unit-Cyclomatic-Complexity, Duplication
- Stability : Test-Quality
- Testability : Unit-Cyclomatic-Complexity, Unit-Size, Test-Quality

To easily compute the avarage a score is represented as a tuple of <int,str> e.g.: <-2,“–”> By doing this we can use one method to compute the avarage score by summing up the int values of the scores and dividing by the count of scores. Afterwards we round the number with the haskell round method and select the appropriate score.

There is also a variable list “scores” included in the Types.rsc file, that contains all the scores as a handy list.

Results

SmallSQL

Metric	Result	Score
Volume	24048 LOCs	++
Unit Complexity	Low: 64% Medium: 13% High: 14% Very High: 9%	–
Unit Size	Low: 71% Medium: 7% High: 11% Very High: 11%	o
Unit Interfacing	Low: 82% Medium: 12% High: 5% Very High: 1%	++
Units	2337	
Duplication	14% (3385 duplicated lines)	-
Testing	42% (973 assert statements)	-

SIG Rating	Score
Maintainability	o
Analysability	o
Changeability	–
Testability	-
Stability	-

HSQLDB

Metric	Result	Score
Volume	167916 LOCs	+
Unit Complexity	Low: 49% Medium: 19% High: 17% Very High: 15%	–

Metric	Result	Score
Unit Size	Low: 55% Medium: 10% High: 12% Very High: 23%	–
Unit Interfacing	Low: 75% Medium: 16% High: 7% Very High: 2%	o
Units	10248	
Duplication	21% (35562 duplicated lines)	–
Testing	6% (631 assert statements)	–

SIG Rating	Score
Maintainability	-
Analysability	-
Changeability	–
Testability	–
Stability	–

Tool usage

To generate a report just import the Main file in the root folder of the source code. Then run the function generateReport(loc location, loc reportFile) * first argument being a eclipse project * second argument where you want the html report to be stored (file has to exist)

The code is structured in 7 files.

Main.rsc

Entry point that contains the generateReport method.

ComplexityAnalyzer.rsc

Contains the AST analyzing. This includes * unit cyclomatic complexity * unit size * unit interfacing * test quality

DuplicationsAnalyzer2.rsc

Contains the duplication analyzing.

VolumeAnalyzer.rsc

Contains the Volume Analyzer for the complete project that is analyzed.

Rater.rsc

Contains all the raters for the individual metrics as well as the avarage function.

Types.rsc

Contains all the custom types that are used in our code. This makes the code more readable and maintainable.

For example to extend the information that is returned for the unit interfacing we only had to change the type once here and where done instead of going through all the code and change the returns, signatures and so on.

Utils.rsc

Contains some helper functions to purify code and count lines.

Tests

For methods with our own types we wrote some basic tests and even found a mistake with that in the avarageScore function. We were not able to come up with a tester for methods that use Rascal M3/AST signatures.

References

[1] I. Heitlager, T. Kuipers, and J. Visser. A Practical Model for Measuring Maintainability. *In Quality of Information and Communications Technology*, 2007. QUATIC 2007. 6th International Conference on the, pages 30–39, Sept 2007. [link].

[2] Jurgen J. Vinju and Michael W. Godfrey. What Does Control Flow Really Look Like? Eyeballing the Cyclomatic Complexity Metric. International Working [link].

[3] Alves, T.L., Correia, J.P., and Visser, J. (2011). Benchmark-based aggregation of metrics to ratings. In Proceedings - Joint Conference of the 21st International Workshop on Software Measurement, IWSM 2011 and the 6th International Conference on Software Process and Product Measurement, MENSURA 2011, pp. 20–29.[link].