Design Decisions
Software Evolution

Arjan Meijer (11425555), Niels Boerkamp (11425547)

November 24, 2017

1 Introduction

During this project we worked on an application which can calculate scores for various properties of source code. The properties we considered are volume, code complexity, code duplication, unit size and unit testing. In this document we will explain the design decisions we made. The calculations and used metrics are based on the Software Improvement Group(SIG) Maintainability model [1]

2 Parsing

To parse the code we create a M3 model and read all the files inside the created model. We also create units by reading all the methods inside the model and removing all the comments from these methods (see Comment Remover. Our units consists of a method name and the (comment free) source-code. We parse the code to a certain format so we can easily apply the metrics to it. The parsing takes up the most time of the calculations of the scores.

3 Comment Remover

The comment remover is designed to read the text only once, the remover reads through all characters in a given string, depending on the last read and current read character, the remover can decide to add the character to a result string. The comment remover also takes strings into account, the comment remover removes: tabs, unnecessary line-endings and multi- & single line comments.

4 Code Duplication

Our approach to the code-duplication problem is based on the 'positional-indexing' of a search engines[2]. The idea is to give every unique line an ID and store the ID of the line that comes next. This gives us the ability to check if there are at least two 'id-streams' that are longer than six lines. A benefit of

this approach over the 'set & map' approach is that our approach is traceable. We have collected the data to point out the duplicated lines in the different files, we do not use this data at the moment because we only calculate the maintainability metrics and do not present the details. The drawback of our approach is that it is slower than the 'set & map' approach. We include lines with one character in our search for duplicates, the result is that we could possibly detect more duplicates. Our justification for this is that we take the end of code blocks into account by using lines with only one character. If we would not do this, we could detect duplicate code that is in reality spread out over multiple code blocks. We only check the contents of the methods for duplicated code. This is in our opinion the most correct way to detect duplications. By only using the contents of the units (methods for java) we skip the imports and package names. This influences the amount of duplicated lines found. A reason for this is that (some) IDE's order the imports alphabetically. This means that, if any two files use the same imports, the imports are counted as duplicates. The same holds for the package name. Therefore, two files that are in the same package and share the first 5 imports will contribute to the duplicated score. These lines are, in our opinion, false positives and should therefore not be counted as duplicated.

5 Unit Size

We calculated the unit size by extracting all the bodies from the methods and counting their lines of code. The metric we applied is the same metric we used for code complexity. We think it is valid to use this metric, since the table contains sensible numbers. A unit with less than ten lines of code is optimal and units with more than fifty lines of code are way to big. Those large methods should result in a lower maintainability score. And small numbers should lead to a higher maintainability score. This metric implies those rules.

6 Test Analyzing metric

The article of the Software Improvement Group[1] suggests two ways of determining the quality and coverage of unit tests. 1) Using an external tool to create a test report, and 2) count the number of 'assert'-statements.

We didn't like the first option, since this approach is very language dependent. According to the SIG should the tool be language independent. The second options wasn't fitting either. We looked at the source code of a Java project and we counted a lot of 'assert'-statements per unit test. Moreover, the number of 'assert'-statements doesn't say anything about the quality of an unit test and about the coverage of those tests.

We came to the conclusion that we had to create a unit test metric ourselves. The implementation we made looks at what percentage of the method names is mentioned in the unit tests. This gives a fairly good representation of the unit

test coverage. Ideally should every method at least be tested once in a unit test.

We have to admit that this metric could provide an inaccurate result. We are currently looking for method names inside the bodies of the unit tests only. For example, if a method calls multiple other methods than they would not be considered as covered by the unit tests.

One could argue that in fact those hidden methods aren't unit tested, since they just happen to be part of a other method. Every smallest piece of code, a unit, should be tested separately That's why it is called unit testing.

For the smallsql-project we calculated a test coverage of 42% and for the bigger one, hsqldb, we calculated a coverage of 24%. This percentage could be lower than the actual test coverage, because of what we explained before.

7 Results

This chapter contains the results of our tool for the smallsql- and hsqldb-project.

7.1 smallsql

Below we present the outcome of our tool. The tool did this analysis in 38 seconds.

Volume	25233 LOC	
Unit Complexity	Low	95.7%
	Moderate	2.4 %
	High	1.6%
	Extreme	0.3%
Code Duplication	Lines	1697
	Percentage	6.7%
Unit size	Low	81.1%
	Moderate	9.3 %
	High	7.4%
	Extreme	2.2%
Unit Test	Test found	181
	Methods found	2219
	Tested methods	935
	Tested percentage	42.1%

Overall score	o
Volume score	++
Unit Complexity Score	-
Code Duplication Score	o
Unit Size Score	-
Unit Testing Score:	-

Analyzability Score	o
Changeability Score	o
Stability Score	-
Testability Score	-

7.2 hsqldb

Below we show the results of the hsqldb-project. The tool ran in 7 minutes.

Volume	171048 LOC	
Unit Complexity	Low	93.5%
	Moderate	4.3 %
	High	1.7%
	Extreme	0.5%
Code Duplication	Lines	20692
	Percentage	12.1%
Unit size	Low	67.4%
	Moderate	16.5 %
	High	11.3%
	Extreme	4.7%
Unit Test	Test found	233
	Methods found	10600
	Tested methods	2513
	Tested percentage	24%

Overall score	o
Volume score	+
Unit Complexity Score	-
Code Duplication Score	-
Unit Size Score	-
Unit Testing Score:	-

Analyzability Score	o
Changeability Score	-
Stability Score	-
Testability Score	-

References

[1] Ilja Heitlager, Tobias Kuipers, and Joost 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. IEEE, 2007.

[2] Iadh Ounis, Gianni Amati, Vassilis Plachouras, Ben He, Craig Macdonald, and Christina Lioma. Terrier: A high performance and scalable information retrieval platform. In *Proceedings of the OSIR Workshop*, pages 18–25, 2006.