Design Decisions
Software Evolution

Arjan Meijer (11425555), Niels Boerkamp (11425547)

November 23, 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]

- parsing
- Comment remover
- line indexing
- test analyzer

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 Remover2). 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 Line indexing

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.

5 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.

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.