Evaluating Human-Assessed Software Maintainability Metrics

Chapter · January 2016

DOI: 10.1007/978-981-10-3482-4_9

CITATIONS
9

READS
281

5 authors, including:

Kamonphop Srisopha
University of Southern California
18 PUBLICATIONS **172** CITATIONS
SEE PROFILE

Lin Shi
Chinese Academy of Sciences
45 PUBLICATIONS **576** CITATIONS
SEE PROFILE

Evaluating Human-Assessed Software Maintainability Metrics

Celia Chen[1], Reem Alfayez[1], Kamonphop Srisopha[1], Lin Shi[2], and Barry Boehm[1]

[1] University of Southern California, 941 Bloomwalk, Los Angeles, CA 90007, USA
{qianqiac, alfayez, srisopha, boehm}@usc.edu
[2] Institute of Software, Chinese Academy of Sciences
shilin@itechs.iscas.ac.cn

Abstract. Being highly maintainable is the key to reduce approximately 75% of most systems' life cycle costs. Software maintainability is defined as the ease with which a software system or a component can be modified, to correct faults, improve performance or other attributes, or adapt to a changed environment. There exist metrics that can help developers measure and analyze the maintainability level of a project objectively. Most of these metrics involve automated analysis of the code. In this paper, we evaluate the software maintainability versus a set of human-evaluation factors used in the Constructive Cost Model II (COCOMO II) Software Understandability (SU) metric, through conducting a controlled experiment on humans assessing SU and performing change-request modifications on open source software (OSS) projects.

Keywords: Software Maintainability, Controlled Experiment, COCOMO II, Open Source Software

1 Introduction

Due to the rapid growth in the demand for software, releasing software fast and using the least amount of resources have become crucial for software companies to survive. In order to acquire those ingredients, software companies have now considered adopting open source software (OSS) as a viable option. However, such adoption is not trivial. With over 18,000 OSS projects available[3], software companies are required to perform careful analysis to ensure that the code is highly maintainable, suits their needs, and interoperates with their other systems with minimal risk and effort. Likewise, an OSS project community needs to ensure high maintainability in order to attract more developers and increase chances of adoption. Strong maintainability challenges may contribute to project abandonment. Understanding what factors affect maintainability is therefore beneficial to both OSS developers and perspective adapters, including assessment of which previously-developed competent to reuse on future projects.

[3] https://sourceforge.net/

Many researchers have been trying to find a way to measure OSS maintainability[10][15][21][23]. Upon reviewing literature, we found that the Maintainability Index (MI) metrics has been widely used and largely studied[6][8][7][17]. However, MI metrics only considers code complexity (Halstead's Volume, McCabe's Cyclomatic Complexity, and source lines of code) and comment ratios as indicators. Due to the nature of OSS and the MI metrics limitation, applying it solely on an OSS project would not give a complete picture of its maintainability. Our current research focuses on identifying complementary methods that could strengthen the ability to evaluate software maintainability.

Boehm's initial Software Qualities (SQs) ontology suggests that maintainability and understandability are closely related[1]. His work on the COCOMO II model also includes human-assessed factors that can measure software understandability[4]. These factors comprise code structure, applications clarity, and self-descriptiveness. In this paper, we conducted a controlled experiment to understand these factors and their relationships with software maintainability in practice.

The remaining paper is organized in the following manner. Section 2 discusses our motivation and research question. Section 3 briefly explains the COCOMO II Software Understandability (SU) factors. We explain the methodology for the controlled experiment and present our initial evaluation results of the COCOMO II factors in Section 4. Some threats to the validity are briefly discussed and highlighted in Section 5. Section 6 concisely mentions works done by other researchers that are overlapping with our research. Section 7 summarizes our findings and some of the planned future work.

2 Research Approach

The principal research question to be addressed by this paper is:

To what extent do COCOMO II SU factors accurately assess to software maintainability?

For this paper, software maintainability of a project is defined as the level of average maintenance effort spent on the project. Less maintenance effort means higher software maintainability. We derived the hypothesis from the above research question.

COCOMO II SU factors Hypothesis: COCOMO II SU factors accurately assess to software maintenance effort.

The corresponding null hypothesis is shown below.

Null Hypothesis: There is no significant relation among SU factors and effort spent on maintenance tasks.

3 COCOMO II Software Understandability factors

The **CO**nstructive **CO**st **MO**del II (COCOMO II) is an objective model for software effort, cost, and schedule estimation[4]. It consists of three main submodels: the early design model, the application composition model, and the

post-architecture model. The post-architecture model is a detailed model that is used once the architecture of the software has been realized. It can be used to predict both software development and maintenance effort. Within the post-architecture model, there exists a sub-model called a reuse model, which suggests that the amount of effort required to modify existing software can be computed through various factors, which includes SU factors[4]. These factors were added to address the maintenance and adaptation underestimates for projects given by the original 1981 COCOMO model, based on data in the studies done by Selby[16], Parikh-Zvegintzov[14], and Gerlich-Denskat[9]. SU factors and their rating scales are shown and explained in Table 1.

Table 1. Rating scale for Software Understanding Increment (SU)

Factor	Very Low	Low	Nominal	High	Very High
Structure	Very low cohesion, high coupling, spaghetti code.	Moderately-low cohesion, high coupling.	Reasonably well-structured; some weak areas.	High cohesion, low coupling.	Strong modularity, information hiding in data/control structures
Application Clarity	No Match between program an application world-views.	Some correlation between program and application.	Moderate correlation between program and application.	Good correlation between program and application.	Clear match between program and application world-views.
Self-Descriptiveness	Obscure code; documentation missing, obscure or obsolete.	Some code commentary and headers; some useful documentation.	Moderate level of code commentary, headers, documentation.	Good code commentary and headers; useful documentation; some weak areas.	Self-descriptive code; documentation up-to-date, well-organized, with design rationale.

4 Initial evaluation of COCOMO II SU factors

4.1 Controlled Experiment Setup

In order to have better control over data and to evaluate the COCOMO II SU factors initially, we conducted a controlled experiment with 11 open source software projects that were maintained by six graduate students of the USC

Computer Science Department. Projects were selected from Sourceforge[4]. The project selection process involved establishing and applying consistent criteria to ensure the quality of this experiment. We excluded projects that are no longer open source or projects that have empty git/cvs/svn repositories. Projects that fall under all of the following criteria were considered:

- The latest stable release is available.
- The size of the source code is relatively reasonable for graduate level students to learn and understand individually.
- The source code is fully accessible.
- The online issue tracking system is active and up-to-date.

Table 2 lists the characteristics of the selected projects. There were more projects that met the above criteria, however they could not all be included into this experiment.

Table 2. Characteristics of project data sources

Language	Number of Projects	Average LOC
Java	6	35,200
PHP	5	67,145

Personal questionnaires were filled out and collected at the beginning of the experiment. Each student reported their industrial experience in a rating from 1 to 5, 1 being extremely inexperienced, 5 being extremely experienced. Over half of the students had at least some levels of industrial experience, including internships and entry-level full-time software engineer jobs at large corporations. One student had experience working in three start-up companies as lead software engineer. However, none of the students had more than five years of industrial experience. Figure 1 shows the distribution of students' industrial experience and Figure 2 shows the experience ratings in details.

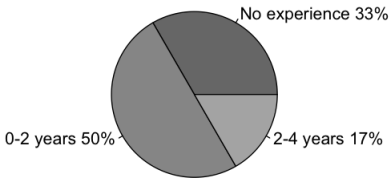

Fig. 1. Pie chart of students' industrial experience

[4] https://sourceforge.net/

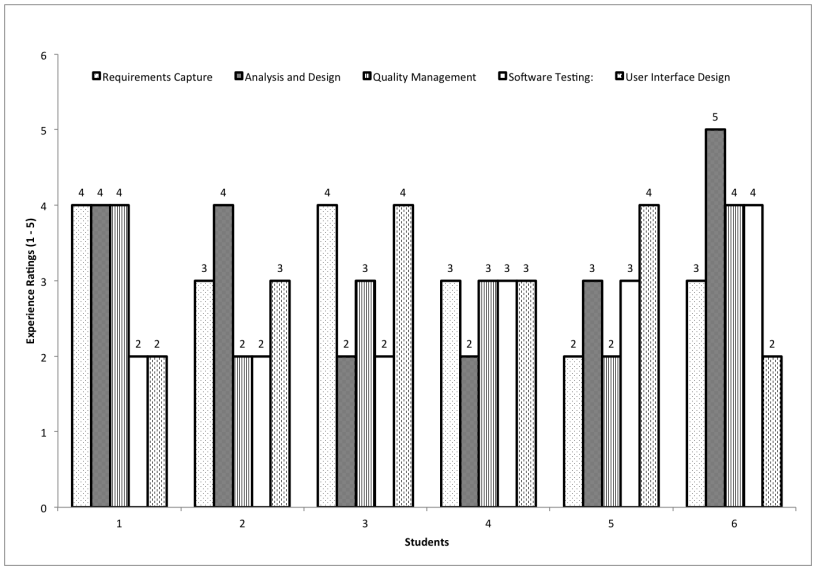

Fig. 2. Experience ratings from students' personnel questionnaire

Students were asked to perform maintenance tasks, including fixing bugs and implementing new feature requests, which were found on each project's corresponding issue tracking website, either Jira[5] or Bugzilla[6]. Each student spent four weeks on one project and one week per task. There were 44 tasks in total. Tasks were assigned to students and a task could be assigned to multiple students. Students were asked to work individually on these tasks. At the end of each week, students were responsible to report efforts spent on the task and answer a questionnaire that consisted a list of questions, which were derived from the COCOMO II SU factors. The answers to those questions were ratings from 1 to 10, 1 being extremely poor and 10 being extremely well. Students were also asked to provide rationale to the ratings they gave to each question. The questions and their corresponding COCOMO II SU factors are as follows:

- Structure: How well are the codes organized? How well are the classes defined in terms of class structure? How well are the variables named? How well are the classes named? Are the classes highly coupled?
- Application Clarity: How well does the software match its application world-views? Are you able to understand the features as described?
- Self-Descriptiveness: How good are the comments? Are there sufficient meaningful comments within the source code? How self-descriptive are the codes? How well is the documentation written? Does the software have sufficient

[5] https://jira.atlassian.com
[6] https://www.bugzilla.org/

documentation to describe interfaces, size, or performance? How well does the current documentation match the current software?

If a student could not finish the assigned task, the student had the option to either continue working on the same task the following week or abandoning the task. Students were asked to only submit the report after finishing a task. Each student had a different total number of assigned projects and tasks based on their availability and experience. Table 3 lists the details of the number of projects and tasks each student worked on.

Table 3. Projects and tasks distribution

Student	1	2	3	4	5	6
Number of Projects	11	5	3	4	5	11
Number of Finished Tasks	35	12	9	7	13	31
Number of Unfinished Tasks	9	8	3	9	7	13
Number of Total Tasks	44	20	12	16	20	44
% of Finished Tasks	79.55%	60.00%	75.00%	43.75%	65.00%	70.45%

Students were required to rate each finished task on a difficulty rating from 1 to 5, 1 being extremely easy and 5 being extremely hard.

4.2 Project Rating Calculation

Once we collected all the above information, we developed an algorithm to calculate the ratings of the COCOMO II SU factors for all projects.

Since students had various levels of experience, in order to keep the consistency of understanding and avoid bias that might be introduced during the experiment, we used their experience ratings and task difficulty ratings as weights when calculating the SU ratings of each project. Since a task could be assigned to multiple students, we used the average ratings from all students who completed the task as the final rating of the task. Each project final SU rating did not come solely from one student, but was the average of all the ratings given by students who worked on that project.

Given a project and a set of maintenance tasks, first we calculated all three SU factors for each task. For each SU factor, we collected the ratings submitted from students who worked on the task. Then we used the task difficulty ratings and student experience ratings as weights to calculate the adjusted SU factor ratings. The final SU rating of a task is the average of all the adjusted SU factor

ratings of the task.

$$Rating_{Task} = \sum_{n=1}^{\infty} StudentRating_n^{Rating_{TaskDifficulty} * Rating_{StudentExperience}} / n \tag{1}$$

After we calculated all three SU factors for each task using the above equation, we calculated the SU factors for the given project by taking the average of SU factors ratings of all the maintenance tasks.

$$Rating_{Project} = \sum_{n=1}^{\infty} Rating_{Task_n} / n \tag{2}$$

Once we repeated the above steps and obtained all the factor ratings for all projects, we performed normalization on the data into a scale between 0 to 10 so that the results are more concise and comparable.

$$NormalizedR_1 = \frac{(Rating_{SUperProject})_1 - \min(Rating_{SUperProject})}{\max(Rating_{SUperProject}) - \min(Rating_{SUperProject})} * 10 \tag{3}$$

$$NormalizedR_i = \frac{(Rating_{SUperProject})_i * NormalizedR_{i-1}}{(Rating_{SUperProject})_{i-1}} \tag{4}$$

where $(Rating_{SUperProject})_1$ is the first not minimum data point in the dataset, $Rating_{SUperProject}$ are all the project level COCOMO II SU factor ratings and $NormalizedR_i$ is the i^{th} normalized data.

4.3 Results and Data Analysis

Collected data points have been analyzed for average effort spent in man-hours per project and the COCOMO II SU factor ratings. Our goal is to test if the COCOMO II SU factors relate to software maintainability. As mentioned in Section 2, we defined software maintainability level as the average effort spent in completing maintenance tasks of a project. Less maintenance effort spent on a project means higher software maintainability. We used Pearson correlation to assess the relationship between each COCOMO II SU factors and average effort. The significance level is set to 0.01, which equals to a confidence level of 99%. Any p-values that is well below that threshold can be concluded as a strong relationship.

Correlation between SU factors and average effort. Table 4 lists the correlation coefficients and their corresponding significance levels.

Table 4. Correlation coefficients matrix between COCOMO II SU factors and average effort spent

Correlation Coefficients Matrix (R values)				
	Average Effort	Application Clarity	Self-Descriptiveness	Structure
Average Effort	1			
Application Clarity	−0.870**	1		
Self-Descriptiveness	−0.929**	0.793*	1	
Structure	−0.940**	0.903**	0.945**	1

*Note *p<.01, **p<.001

There was a strong negative correlation between average effort and Structure, R = -0.94013, n = 11, p = 0.00002. Higher quality structure was correlated with less effort spent on maintenance tasks hence higher maintainability. There was a strong negative correlation between average effort and Application Clarity, R = -0.87032, n = 11, p = 0.00049. When software reflects higher application content clarity, developers spent less effort on maintenance tasks hence higher maintainability. There was a strong negative correlation between average effort and Self-Descriptiveness, R = -0.92996, n = 11, p = 0.00003. More self-descriptive source code was correlated with less effort spent on maintenance tasks hence higher maintainability.

Correlation between factors within Structure and average effort. Table 5 lists the correlation coefficients and their corresponding significance levels.

Table 5. Correlation coefficients matrix between factors within Structure and average effort spent

Correlation Coefficients Matrix (R values)						
	Average Effort	Code Organization	Class Names	Variable Names	Class Structure	Coupling/ Cohesion
Average Effort	1					
Code Organization	−0.916**	1				
Class Names	−0.876**	0.881**	1			
Variable Names	−0.879**	0.884**	0.962**	1		
Class Structure	−0.836*	0.777*	0.742*	0.737*	1	
Coupling/Cohesion	−0.848**	0.822*	0.792*	0.782*	0.849**	1

*Note *p<.01, **p<.001

There was a strong negative correlation between average effort and Code Organization, R = -0.91552, n = 11, p = 0.00008. Higher quality code organization

was correlated with less effort spent on maintenance tasks hence higher maintainability. There was a strong negative correlation between average effort and Class Names, R = -0.8762, n = 11, p = 0.0004. More meaningful and highly content-reflective class names were correlated with less effort on maintenance tasks hence higher maintainability. There was a strong negative correlation between average effort and Variable Names, R = -0.87884, n = 11, p = 0.00037. More meaningful and highly content-reflective variable names were correlated with less effort on maintenance tasks hence higher maintainability. There was a strong negative correlation between average effort and Class Structure, R = -0.83562, n = 11, p = 0.00136. Better structured and designed classes were correlated with less effort on maintenance tasks hence higher maintainability. There was a strong negative correlation between average effort and Coupling/Cohesion, R = -0.84778, n = 11, p = 0.00098. Lower coupling and higher cohesion was correlated with less effort on maintenance tasks hence higher maintainability.

Correlation between factors within Self-Descriptiveness and average effort. Table 6 lists the correlation coefficients and their corresponding significance levels.

Table 6. Correlation coefficients matrix between factors within Self-Descriptiveness and average effort spent

Correlation Coefficients Matrix (R values)				
	Average Effort	Self-Descriptive Code	Code Commentary	Documentation Quality
Average Effort	1			
Self-Descriptive Code	−0.921**	1		
Code Commentary	−0.952**	0.984**	1	
Documentation Quality	0.039	−0.064	0.024	1

*Note *p<.01, **p<.001

There was a strong negative correlation between average effort and Self-descriptive Code, R = -0.92139, n = 11, p = 0.00006. More self-descriptive code was correlated with less effort spent on maintenance tasks hence higher maintainability. There was a strong negative correlation between average effort and Code Commentary, R = -0.95191, n = 11, p = 6.40660E-6. Higher quality comments and comments density was correlated with less effort on maintenance tasks hence higher maintainability. However, there was no correlation between average effort and Documentation Quality, R = 0.03936, n = 11, p = 0.90853. Documentation quality was not correlated with effort spent on maintenance tasks, thus confirming the null hypothesis in this case. This is a partial refutation of the possibility that subjects will always give higher low maintainability aspect assessments to systems they found easier or harder to maintain.

In conclusion, the results indicate that the strength of association between each COCOMO II SU factor and average maintenance effort is very high, and that the correlation coefficient is very highly significant. Therefore, COCOMO II SU factors are significantly related to software maintainability. Taken together, these results suggest that COCOMO II SU factors have strong negative association with software maintenance effort, hence strong positive association with software maintainability.

5 Threats to validity

External validity: 11 projects were studied for this study, which might limit the generalizability of the results. Also, there may be differences between open-source, outsourced, and in-house performance. In order to mitigate this threat, we have been recruiting more developers and adding new projects to this study.

The maintenance activities we performed in this study were mainly bug fixes and feature request implementation. However, there are lots of other possible maintenance activities that we did not cover in this study. It may include change accommodation to input data and operating systems, code efficiency improvement, and other activities. The classic 1980 Lientz-Swanson survey of 487 business data process organizations[11] indicated that 21.7% of their software maintenance activity was in corrective maintenance (Emergency program fixes and routine debugging) and 41.8% of the software maintenance effort was in software enhancement along with 23.6% due to adapting to changes in data, files, hardware, and operating systems; and 12.9% in other effort such as improving documentation and efficiency. Thus, 63.5% of the overall code-related effort was in the code enhancement and correction activities in this study. Later data is generally similar, but may vary by domain. Also, a good deal of a maintenance organizations effort is devoted to business management functions, although its level is often roughly proportional to the code-maintenance effort.

Internal validity: Although the participants were all graduate level students, most of them none to moderate professional experience. In order to mitigate this threat for future results, we have been recruiting students with more professional experience and experience working on open-source projects.

The accuracy of questionnaire responses by the students is somewhat questionable. One case is that some students may not be recording the number of hours they spent on each task when they work. When they were reminded to submit the questionnaire, they filled the form out with best-guess hours. The threat is mitigated by having them install time tracking plug-ins (e.g. WakaTime[7]) on their IDEs so that effort spent on each task can be recorded. Another aspect is that some students may randomly give ratings to the questions. This threat is mitigated by asking them to provide rationale to the ratings.

[7] https://marketplace.eclipse.org/content/wakatime

6 Related work

Metrics to Evaluate Maintainability. Oman and Hagemeister introduced a composite metric for quantifying software maintainability[13]. This Maintainability Index (MI) has evolved into numerous variants and has been applied to a number of industrial software systems. Coleman et al. defined the maintainability index as a function of Average Halstead Volume (V) per Module, Average Cyclomatic Complexity per Module, Average Lines of Code per Module and Average Percent of Lines of Code per Module[7]. Welker revised the MI proposed by Oman and Hagemeister by reducing the emphasis in comments to improve maintainability[17].

Maintainability-related Factors. Yamashita and Moonen conducted empirical studies on the relation between code smells and maintainability[18][19][20]. They investigated the extent to which code smells reflect factors affecting maintainability that have been identified as important by programmers. They found that some of the factors can potentially be evaluated by using some of the current code smell definitions, but not all of the maintainability aspects that were considered important by professional developers. They also observed that certain inter-smell relations were associated with problems during maintenance and also that some inter-smell relations manifested across coupled artifacts.

Chen et al. conducted empirical studies on the Maintainbility aspects, including not only literature and model analyses, but also data-analytics studies of the maintainability index of open-source software artifacts[6].

Quantifying Maintainability. Zhang et al. presented an automated learning-based approach to train maintainability predictors by harvesting the actual average maintenance effort computed from the code change history[22]. Their evaluation showed that SMPLearner outperformed the traditional 4-metric MI model and also the recent learning-based maintainability predictors constructed based on single Class-level metrics. They observed that single Class-level metrics were not sufficient for maintainability prediction.

Boehm et al. provided quantitative information to help projects determine how maintenance cost and duration vary as a function of which Maintainability methods, processes, and tools (MPTs) to use in which situations[1]. They have developed various partial solutions such as lists of Maintainability-enhancing practices[2][3]; cost model drivers that increase cost to develop but decrease cost to maintain such as required reliability and architecture and risk resolution[4]; MPTs that reduce life cycle technical debt[5]; and quantification of early-increment maintenance costs as sources of later-increment productivity decline in incremental and evolutionary development[12].

7 Conclusions and future work

This paper has described the analysis of how human-assessed COCOMO II SU factors relate to software maintainability. OSS projects were studied through a

controlled experiment. The results were found to show with statistical significance that the COCOMO II SU factors are highly related to software maintainability - the COCOMO II SU factors have strong negative association with software maintenance effort, hence strong positive association with software maintainability. Further work will have to be done on more OSS projects as well as closed source projects. We plan to expand our study by increasing the number of projects and recruiting senior developers with more experience. In addition, other maintainability enablers in software architecture and V&V support need to be studied, such as Diagnosability, Accessibility and Testability. We are also working on comparing the human-assessed maintainability metrics with their automated maintainability and technical debt assessment counterparts.

References

1. Boehm, B., Chen, C., Srisopha, K., Shi, L.: The key roles of maintainability in an ontology for system qualities. In: 26th Annual INCOSE International Symposium (2016)
2. Boehm, B.W.: Software and its impact: A quantitative assessment. Datamation pp. 48–59 (1973)
3. Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative evaluation of software quality. In: Proceedings of the 2nd international conference on Software engineering. pp. 592–605. IEEE Computer Society Press (1976)
4. Boehm, B.W., Madachy, R., Steece, B., et al.: Software cost estimation with Cocomo II with Cdrom. Prentice Hall PTR (2000)
5. Boehm, B.W., Valerdi, R., Honour, E.: The ROI of systems engineering: Some quantitative results for software-intensive systems. Systems Engineering 11(3), 221–234 (2008)
6. Chen, C., Shi, L., Srisopha, K.: How does software maintainability vary by domain and programming language? In: The 27th Annual IEEE Software Technology Conference
7. Coleman, D., Ash, D., Lowther, B., Oman, P.: Using metrics to evaluate software system maintainability. Computer 27(8), 44–49 (1994)
8. Ganpati, A., Kalia, A., Singh, H.: A comparative study of maintainability index of open source software. Int. J. Emerg. Technol. Adv. Eng 2, 228–230 (2012)
9. Gerlich, R., Denskat, U.: A cost estimation model for maintenance and high reuse. In: Proceedings, ESCOM (1994)
10. Ghosheh, E., Black, S., Qaddour, J.: Design metrics for web application maintainability measurement. In: 2008 IEEE/ACS International Conference on Computer Systems and Applications. pp. 778–784. IEEE (2008)
11. Lientz, B.P., Swanson, E.B.: Software maintenance management (1980)
12. Moazeni, R., Link, D., Boehm, B.W.: Incremental development productivity decline. In: 9th International Conference on Predictive Models in Software Engineering, PROMISE '13, Baltimore, MD, USA, October 9, 2013. pp. 7:1–7:9 (2013)
13. Oman, P., Hagemeister, J.: Metrics for assessing a software system's maintainability. In: Software Maintenance, 1992. Proceedings., Conference on. pp. 337–344. IEEE (1992)
14. Parikh, G., Zvegintzov, N.: The world of software maintenance. Tutorial on software maintenance pp. 1–3 (1983)

15. Samoladas, I., Gousios, G., Spinellis, D., Stamelos, I.: The sqo-oss quality model: measurement based open source software evaluation. In: IFIP International Conference on Open Source Systems. pp. 237–248. Springer (2008)
16. Selby, R.W.: Empirically analyzing software reuse in a production environment. In: Software Reuse: Emerging Technology. pp. 176–189. W. Tracz (Ed.), IEEE Computer Society Press (1988)
17. Welker, K.D.: The software maintainability index revisited. CrossTalk 14, 18–21 (2001)
18. Yamashita, A., Moonen, L.: Do code smells reflect important maintainability aspects? In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on. pp. 306–315. IEEE (2012)
19. Yamashita, A., Moonen, L.: Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: Proceedings of the 2013 International Conference on Software Engineering. pp. 682–691. IEEE Press (2013)
20. Yamashita, A., Moonen, L.: To what extent can maintenance problems be predicted by code smell detection? an empirical study. Information and Software Technology 55(12), 2223–2242 (2013)
21. Yu, L., Schach, S.R., Chen, K.: Measuring the maintainability of open-source software. In: 2005 International Symposium on Empirical Software Engineering, 2005. pp. 7–pp. IEEE (2005)
22. Zhang, W., Huang, L., Ng, V., Ge, J.: Smplearner: learning to predict software maintainability. Automated Software Engineering 22(1), 111–141 (2015)
23. Zhou, Y., Xu, B.: Predicting the maintainability of open source software using design metrics. Wuhan University Journal of Natural Sciences 13(1), 14–20 (2008)