Questioning software maintenance metrics: A comparative case study

Conference Paper · September 2012

DOI: 10.1145/2372251.2372269

Dag I. K. Sjøberg
University of Oslo
132 PUBLICATIONS **8,567** CITATIONS

Bente Anda
University of Oslo
40 PUBLICATIONS **2,385** CITATIONS

Questioning Software Maintenance Metrics: A Comparative Case Study

Dag I.K. Sjøberg
Department of Informatics,
University of Oslo, P.O. Box 1080
Blindern, NO-0316 Oslo, Norway
Dag.Sjoberg@ifi.uio.no

Bente Anda
Department of Informatics,
University of Oslo, P.O. Box 1080
Blindern, NO-0316 Oslo, Norway
Bente.Anda@ifi.uio.no

Audris Mockus
Department of Software,
Avaya Labs Research,
Basking Ridge, NJ 07920. USA
audris@avaya.com

ABSTRACT
Context: Many metrics are used in software engineering research as surrogates for maintainability of software systems. **Aim**: Our aim was to investigate whether such metrics are consistent among themselves and the extent to which they predict maintenance effort at the entire system level. **Method**: The Maintainability Index, a set of structural measures, two code smells (Feature Envy and God Class) and size were applied to a set of four functionally equivalent systems. The metrics were compared with each other and with the outcome of a study in which six developers were hired to perform three maintenance tasks on the same systems. **Results**: The metrics were not mutually consistent. Only system size and low cohesion were strongly associated with increased maintenance effort. **Conclusion**: Apart from size, surrogate maintainability measures may not reflect future maintenance effort. Surrogates need to be evaluated in the contexts for which they will be used. While traditional metrics are used to identify problematic areas in the code, the improvements of the worst areas may, inadvertently, lead to more problems for the entire system. Our results suggest that local improvements should be accompanied by an evaluation at the system level.

Categories and Subject Descriptors
D.2 SOFTWARE ENGINEERING

General Terms
Measurement, Experimentation

Keywords
Software maintenance, software metrics

1. INTRODUCTION
It is well known that software maintenance is costly and effort intensive. Therefore, software systems should be maintainable. However, how do we know which systems will be maintainable? What designs and implementations of a given set of requirements would be most maintainable? How can source code be improved to make it more maintainable?

To help answer such questions, much of software engineering research over the years has been devoted to *software maintenance metrics*. Examples are the Maintainability Index [16], the CK metrics, including coupling and cohesion [5] and various code smells [9].

Previous research on the validation of these metrics and approaches has investigated systems that were functionally different. Differences in functionality make it difficult to isolate the effects of design choices from the functionality of the systems. In contrast, we are in a unique situation in that we have access to four industry-quality systems that are functionally equivalent. As a part of an investigation on the trade-offs between the costs of developing the systems and quality improvement, we assessed the maintainability of the four systems by using the metrics described above and the results of a particular maintenance study of these systems with six developers from two companies. In summary, the research questions of the study reported in this paper are as follows:

RQ1: Are commonly used software maintainability metrics mutually consistent at the system level?

RQ2: Are commonly used software maintainability metrics related to the actual maintenance effort observed in our study?

The remainder of this paper is organized as follows. Section 2 describes the four systems that are being the objects of this comparative study. Section 3 describes the maintenance metrics that were applied to the systems. Section 4 describes the results of the maintenance study on the systems. Section 5 discusses the results reported in the previous sections. Section 6 concludes.

2. THE FOUR SYSTEMS
The four systems available in this comparative case study were functionally equivalent (with the same requirements specifications) web-based information systems primarily implemented in Java. They were developed independently by four different companies at the costs of €18,000, €25,000, €52,000 and €61,000. The sizes of the four systems, named Systems A through D, are shown in the upper two rows of Table 1. The systems were developed as part of a study on the variability and reproducibility in software engineering [2].

3. SOFTWARE MAINTENANCE METRICS
This section describes the set of metrics that were selected because they are among the most used and well known.

3.1 Maintainability Index
The Maintainability Index (MI) has been proposed for assessing the maintainability of complete systems. The original three-metric MI uses a polynomial to combine the average per module of three traditional code measures (lines of code, cyclomatic complexity

and Halstead Volume) into a single-value indicator of maintainability [16]. An improved four-metrics version of MI also includes the number of comments.

In conventional non-object-oriented systems, the values of the improved MI have been classified as follows: >85 indicate good maintainability; 65-85 indicate moderate maintainability; and 65 and below indicate poor maintainability with very poor pieces of code (big, uncommented, unstructured) [16].

To our knowledge, there are no heuristics for MI classification values for object-oriented systems. However, because classes are smaller in such systems than modules in conventional systems, researchers have argued that the thresholds for object-oriented systems should be higher [17]. This observation is consistent with the values that we found for our four systems. Table 1 shows that the MI values range from 113 (System A) to 120 (System D).

3.2 Structural Measures

The most common set of metrics for assessing code maintainability is structural measures (SM), including the CK metrics [5]. A previous study [3] used an adapted version of a subset of the CK metrics to evaluate the four systems that are also the subject of this study. The subset includes the coupling measure OMMIC (call to methods in an unrelated class), the cohesion measure TCC (tight class cohesion) and the measure of size of classes WMC1 (number of methods per class; each method has a weight of 1) and depth of inheritance tree (DIT). The values of the respective systems are shown in Table 1. By combining these and a few more metrics, the previous study ranked System D as the most maintainable one, System A slightly ahead of System B, and System C as the least maintainable system [3].

3.3 Code Smells

The concept of code smells was introduced as an indicator of problems with the software design [9]. Code smells have become an established way of indicating issues with software designs that may cause problems for future development and maintenance [9], [12]. However, a systematic review [18] found only five studies that investigated the impact of code smells on maintenance.

Nevertheless, we recently conducted a study [14] on the effects of 12 code smells on the maintenance effort in the four systems that are the subject of the study reported here. We found that only two smells negatively affected the maintenance effort (Feature Envy and God Class). However, this result was derived before we had adjusted for the file size and the number of changes. When we adjusted for these aspects, we found no effect. Still, to illustrate the use of code smells, we included the average number per KLOC of these two smells in the four systems in Table 1.

Note that the paper [14] identified the effects of various code smells on maintenance effort by analyzing the number of smells and the maintenance effort at the file level. We did not study the variation among the systems *per se*. Neither did we study whether refactoring of files to reduce smells, which may lead to reductions in size of the individual files, might increase the total system size and thus not improve the overall system maintainability. In contrast, the study reported here operates at the system level.

4. MAINTENANCE STUDY

Several maintenance metrics that were applied to the four systems were described above. However, how well do these metrics indicate actual maintainability (how easy it is to maintain the systems in practice)?

To find out, we conducted a controlled maintenance study on the four systems. We hired six developers for a total cost of €50,000 to perform three maintenance tasks each on two of the four functionally equivalent but independently developed Java systems. Three of the developers worked for a software company in the Czech Republic, and another set of three developers worked for a software company in Poland. We recruited the developers from a pool of 65 participants in an earlier study on programming skill [4] that also included maintenance tasks. Based on the results of that study, we selected these six developers because they could program reliably at medium to high levels of performance, reported high levels of motivation to participate in the study and were available to take part in new studies. In this case, the results of the former study became the pre-test measures for our study. In general, using pre-test measures to maximize the interpretability of the results is recommended [11].

The developers implemented two adaptive tasks that were needed to allow the systems to become operational again after changes had been made to the web platform. The developers also implemented a third task that was requested by the users. The amount of time that each developer spent on each file was automatically recorded by a plug-in to an Eclipse IDE. The study lasted three to four weeks in the Czech Republic and three weeks in Poland.

Each developer conducted the same three tasks on two different systems. There are two reasons for having the same developer maintain two systems. First, the relative impact of a system can be separated from the impact of the developer. Second, we could observe the developers' learning process when they implemented the same tasks the second time. These two rounds also correspond to two different settings commonly found in maintenance work: maintainers who are newcomers to a system and maintainers who are already familiar with it. Although the systems were assigned randomly to each developer, the four systems were maintained the same number of times (two) by each developer, and all of the systems were maintained at least once in each round.

The next lowest row of Table 1 shows the average amount of time that the developers spent on each of the systems. On average, the developers spent 39% less time on performing the tasks in the second round. We adjusted for this difference in the calculation of the average values in Table 1.

Usually in human-centric studies in software engineering, one measures the quality of the tasks performed by the subjects in addition to time (effort). Perhaps the most commonly used quality attribute is the number of defects. However, in our case, the acceptance tests showed that there were few defects in the systems after the maintenance tasks had been performed. Therefore, it was not meaningful to use defects as a quality indicator. Instead, we used the number of changes completed in the course of the task as an indicator of quality. The number of changes is typically found to be a good predictor of later defects, with more changes increasing the fault-proneness [10], [8]. Consequently, we also included the number of changes (revisions) performed to implement the tasks as the last control variable. The numbers were calculated using SVNKit [15], which is a Java library for obtaining information from Subversion the Subversion version control system.

The last row of Table 1 shows the average number of revisions per system. By combining the scores for effort and quality, Table 1 shows that System C is the most maintainable system. In contrast, System B has the lowest quality, and System A had the highest maintenance effort.

Table 1. Maintenance metrics and a maintenance study applied to the four systems
Legend: green indicates the best system and red the worst one

Category	Metrics	System A	System B	System C	System D
Size	Number of Java files	63	**168**	**29**	119
	Java lines of code (LOC)	8205	**26679**	**4983**	9960
Maintainability Index (MI)	LOC, # of comments, cyclomatic complexity, Halstead's volume	**113**	117	114	**120**
Structural measures (SM)	Coupling (OMMIC)	7.7	5.3	**8.6**	**4.7**
	Cohesion (TCC)	**0.26**	0.17	0.20	**0.11**
	Size of classes (WMC1)	6.9	7.8	**11.4**	**4.9**
	Depth of inheritance tree (DIT)	0.46	0.75	**0**	**0.83**
Code smells	Feature Envy (# per KLOC of code)	**4.51**	**1.27**	3.41	2.51
	God Class (# per KLOC of code)	**0.12**	0.19	**0.60**	0.20
Study of actual maintainability	Average effort (hours)	18	**33**	**13**	23
	Predictor of quality (avg. # changes)	**148**	125	**76**	124

5. COMPARISON AND DISCUSSION

Table 1 shows that none of the tailored maintenance metrics ranked the system that performed best in the maintenance study (System C) as the best one.

The Spearman rank correlations among the maintainability metrics and the observed maintenance effort and quality are shown in Table 2. High values of metrics MI, TCC and DIT are supposed to indicate high maintainability. (Note that higher DIT values are considered good at least up to three [6]. The DIT values in our study vary from 0 to 0.83 on average.) Therefore, to compare the variables directly with maintainability, expressed in the number of hours spent on the maintenance tasks, we inverted the MI, TCC and DIT.

The only two metrics that are highly correlated with effort are size and the inverse of cohesion (1/TCC). The remaining maintainability surrogate metrics are negatively correlated with the observed effort. Three of them, 1/MI, OMMIC and FE, have a high negative correlation.

The inconsistency among the metrics is striking. One reason for the inconsistency is that some metrics are strongly interdependent; that is, for a given system, improving the value of one metric may imply less favourable values for other ones. For example, achieving low coupling is more difficult if one also attempts to achieve high cohesion, and vice versa. Achieving high cohesion and low coupling among modules or classes would generally be easier if one were to increase their size, but then the overall maintainability would decrease because of the increase in the size of the module or class.

Similarly, some practices, such as the refactoring of God Classes, may lead to more files. Although this practice may decrease the size of what was originally a God-Class file, it would lead to a larger system overall. Thus, such apparent improvements induced by reducing the size of God Classes may make it more difficult to maintain the new refactored system.

The conformance between code size and the outcome of the maintenance study may not be surprising. Software engineering folklore states that reducing functionality will reduce maintenence problems. The systems in our study demonstrate the positive effect of reducing size (measured in number of files or total LOC) without reductions in functionality.

The answers to our research questions are as follows:

RQ1: The considered common maintainability metrics were not mutually consistent in the considered projects.

RQ2: Among the considered maintainability metrics, only size and the inverse of cohesion were strongly correlated with the actual maintenance effort observed in the study.

It is possible that metrics apart from size may play a role in reducing maintenance effort in large projects where it takes a long time (> 3 years) for developers to become fluent [19], but we see no evidence that they matter in our context.

Table 2. Spearman rank correlation matrix

	LOC	1/MI	OMMIC	1/TCC	WMC1	1/DIT	FE	GC	Hours	# chgs
LOC	1	-0.6	-0.8	0.6	-0.4	0.6	-0.8	-0.4	1	0.4
1/MI	-0.6	1	0.8	-1	0.4	1	0.8	-0.4	-0.6	0.4
OMMIC	-0.8	0.8	1	-0.8	0.8	1	0.6	0.2	-0.8	-0.2
1/TCC	0.6	-1	-0.8	1	-0.4	-1	-0.8	0.4	0.6	-0.4
WMC1	-0.4	0.4	0.8	-0.4	1	0.5	0	0.4	-0.4	-0.4
1/DIT	-0.5	1	1	-1	0.5	1	0.5	-1	-0.5	1
FE	-0.8	0.8	0.6	-0.8	0	0.5	1	-0.2	-0.8	0.2
GC	-0.4	-0.4	0.2	0.4	0.4	-1	-0.2	1	-0.4	-1
Hours	1	-0.6	-0.8	0.6	-0.4	-0.5	-0.8	-0.4	1	0.4
# chgs	0.4	0.4	-0.2	-0.4	-0.4	1	0.2	-1	0.4	1

Even though the study controlled for functionality and other factors, having only four sample points makes it difficult to make sweeping generalizations. The inherent inconsistency of maintainability metrics and the strength of the anti-correlations with observed maintainability, strongly suggest that, at least in the context of smaller scale projects, the size of the system may be the decisive factor determining actual maintainability.

6. CONCLUSIONS

The results of this comparative case study indicate that the existing software maintenance metrics are mutually inconsistent and that none of them, apart from size and the inverse of cohesion, are consistent with the results of a maintenance study of four systems that implemented the same functionality. Still, these metrics are used to validate a great number of technologies (processes, methods, techniques and tools) for supporting software maintenance. The choice of metrics, rather than actual maintainability, may determine the outcome of a study. For example, using the Maintainability Index, one could argue that hiring an expensive company with heavy processes (System D) improves maintainability. On the other hand, using size, one could claim that it is better to hire an inexpensive company with light processes (System C) because it will produce smaller systems that, consequently, are more maintainable.

Our results are consistent with the findings of a related systematic literature review [13]. The researchers of this study found that there was little evidence regarding the effectiveness of software maintainability prediction techniques and models.

Our study controlled for a number of factors including functionality, application domains and programming language, and the authors did not develop any of the metrics (lack of experimenter bias). Our contribution is the observation that at the entire system level, the simplest metric of size was the best predictor of maintainability.

Consequently, this study indicates that overall system size (as opposed to, e.g., file size or class size) as a measure of maintainability has been underrated in the software engineering community. However, the other “sophisticated” maintenance metrics are overrated. Researchers in software engineering should be cautious when using such metrics as surrogates for actual maintainability unless the metrics have been properly evaluated in the same context for which they serve as surrogates.

ACKNOWLEDGMENTS

We thank Aiko Yamashita for providing support with the data collection. This work was partly funded by the Research Council of Norway through the projects AGILE, grant 179851/I40, and TeamIT, grant 193236/I40.

REFERENCES

[1] B. Anda. Assessing Software System Maintainability using Structural Measures and Expert Assessments, Proc. 23rd Int’l Conf. on Software Maintenance, pp. 204-213, 2007.

[2] B.C.D. Anda, D.I.K. Sjøberg and A. Mockus. Variability and Reproducibility in Software Engineering: A Study of Four Companies that Developed the Same System, *IEEE Trans. Softw. Eng*, vol. 35, no. 3, pp. 407–429, 2009.

[3] H.C. Benestad, B. Anda and E. Arisholm. Assessing Software Product Maintainability Based on Class-Level Structural Measures, Proc. 7th Int’l Conf. on Product-focused Software Process Improvement, LNCS 3009, Springer-Verlag, pp. 94-111, 2006.

[4] G.R. Bergersen and J.E. Gustafsson, Programming Skill, Knowledge and Working Memory Among Professional Software Developers from an Investment Theory Perspective, *J. Individual Differences*, vol. 32, no. 4, pp. 201-209, 2011.

[5] S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object Oriented Design, *IEEE Trans. Softw. Eng*., vol. 20, no. 6, pp. 476–493, 1994.

[6] J. Daly, A. Brooks, J. Miller, M. Roper and M. Wood. An Empirical Study Evaluating Depth of Inheritance on the Maintainability of Object-Oriented Software, *Empirical Softw. Eng*., vol. 1, pp. 109-132, 1996.

[7] D. Darcy and C.F. Kemerer. OO Metrics in Practice, *IEEE Software*, vol. 22, no. 6, Nov./Dec. 2005, pp. 17–19, 2005.

[8] S.G. Eick, T L. Graves, A. F. Karr, J.S. Marron and A. Mockus, Does code decay? Assessing the evidence from change management data, *IEEE Trans. on Softw. Eng*., vol. 27, no. 7, pp. 1-12, 2001.

[9] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[10] T. Hall, S. Beecham, D. Bowes, D. Gray and S. Counsell. A Systematic Review of Fault Prediction Performance in Software Engineering, *IEEE Trans. Softw. Eng*., 2011 (preprint).

[11] V.B. Kampenes, T. Dybå, J.E. Hannay and D.I.K. Sjøberg, A Systematic Review of Quasi-Experiments in Software Engineering. *Inf. and Softw. Tech*., vol. 51, pp. 71-82, 2009

[12] M. Lanza, R. Marinescu and S. Ducasse, Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems. Springer-Verlag New York, Inc, 2005.

[13] M. Riaz, M. Mendes and E.D. Tempero. A Systematic Review of Software Maintainability Prediction and Metrics. 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM 2009), Lake Buena Vista, FL, USA, 15-16 Oct. 2009, pp. 367-377, 2009.

[14] D.I.K. Sjøberg, A. Yamashita, B. Anda, A. Mockus and T. Dybå. Quantifying the Effect of Code Smells on Maintenance Effort. Submitted for publication in *IEEE Trans. Softw. Eng*. 2012.

[15] TMate-Sofware. SVNKit - Subversioning for Java. [Cited June 2010]; Available from: http://svnkit.com/.

[16] K.D. Welker, P.W. Oman and G.G. Atkinson. Development and Application of an Automated Source Code Maintainability Index. *Software Maintenance: Research and Practice,* vol. 9, pp. 127-159, 1997.

[17] K.D. Welker. The Software Maintainability Index Revisited, *CrossTalk*, August 2001.

[18] M. Zhang, T. Hall and N. Baddoo, Code Bad Smells: A Review of Current Knowledge, *Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179–202, 2011.

[19] M. Zhou and A. Mockus. Developer fluency: Achieving True Mastery in Software Projects. ACM SIGSOFT / FSE, pp. 137-146, Santa Fe, New Mexico, Nov. 7-11, 2010.