Maintainability Index Revisited
- position paper -

Tobias Kuipers
Software Improvement Group, Amsterdam, The Netherlands, t.kuipers@sig.nl
Joost Visser
Universidade do Minho, Braga, Portugal, joost.visser@di.uminho.pt

Abstract – **The amount of effort needed to maintain a software system is related to the technical quality of the source code of a system. In the past, a Maintainability Index has been proposed to calculate a single number that determines the maintainability of a system. The authors have identified some problems with that index, and propose a Maintainability Model which alleviates most of these problems.**

INTRODUCTION

In [1] Oman et. al proposed the Maintainability Index: an attempt to objectively determine the maintainability of software systems based upon the status of the source code. This measure is based on measurements the authors performed on a number of software systems and calibrating these results with the opinions of the engineers that maintained the systems. The results for the systems examined by Oman et. al were plotted, and a fitting function was derived. The resulting fitting function was then promoted to be the Maintainability Index producing function. Subsequently, a small number of improvements were made to the function.

We have used the Maintainability Index in our consultancy practice over the last four years, and found a number of problems with it. Although we see a clear use for determining the maintainability of the source code of a system in one (or a few) simple to understand metrics, we have a hard time using the Maintainability Index to the desired effect. After a discussion of aspects of the MI we find make it hard to use, we will proceed to present a preliminary set of metrics we feel serve the same goal as the MI, but do suffer from its problems less.

MAINTAINABILITY INDEX

The MI is a composite number, based on several unrelated metrics for a software system. It is based on the Halstead Volume (HV) metric [2], the Cyclomatic Complexity (CC) [3] metric, the average number of lines of code per module (LOC), and optionally the percentage of comment lines per module (COM). Halstead Volume, in turn, is a composite metric based on the number of (distinct) operators and operands in source code. The complete fitting function is 171 - 5.2 * ln(HV) - 0.23 * CC - 16.2 * ln (LOC) + 50 * sin (sqrt(2.4 * COM)). The higher the MI, the more maintainable a system is deemed to be.

Root-cause analysis

Since the MI is a composite number, it is very hard to determine what causes a particular value for the MI. The acceptance of a numerical metric with practitioners, we find, increases greatly when they can determine what change in a system caused a change in the metric. When the MI has a particularly low value, it is not immediately clear how to increase it.

Average complexity

One of the metrics used to compose the MI is the average Cyclomatic Complexity. We feel this is a fundamentally flawed number. Particularly for systems built using OO technology, the complexity per module will follow a power law distribution. The average complexity will invariably be low (e.g. because all setters and getters of a Java system have a complexity of 1), whereas anecdotal evidence suggests that the maintenance problems will occur in the few outliers that have a complexity of over 100.

Computability

Particularly the Halstead Volume metric is difficult to compute. There is no formal definition of what constitutes an operator or an operand in a language such as Java or C# [4]. Halstead Volume is a metric that is not widely accepted within the software engineering community. See [5] for a critique.

Comment

The implication of using the number of lines of comment as a metric is that a “well documented” piece of code is better to maintain than a piece of code that is not documented at all. Although this appears to be a logical notion, we find that counting the number of lines of comment, in general, has no relation with maintainability whatsoever. More often than not, comment is simply code that has been “commented out”, and even if it is natural language text it sometimes refers to earlier versions of the code. Apparently, the authors of the MI had reservations about measuring comment, as they made this part of the MI optional.

Understandability

Using the MI proves to be hard, both on the management level as well as on the technical/developer level. We find that the lack of control the developers feel they have over the value of the MI makes them dismissive of the MI for quality assessment purposes. This directly influences management acceptance of the value. Although having a measure such as the MI at your disposal is obviously more useful than knowing nothing about the state of your systems, the lack of "knobs to turn" to influence the value makes it less useful as a management tool.

TOWARDS THE SIG MAINTAINABILITY MODEL

To alleviate (some) of the problems we discussed above, we are in the process of establishing the SIG Maintainability Model. This is by no means a finished or even mature model, but work in progress. Here, we would like to share our initial findings, and welcome feedback from the academic community.

The model currently consists of five easy to calculate metrics that are not composed, but merely used in concert. Each of the five metrics can be fairly easily calculated, and can be easily defined. Root causes of anomalous numbers can be easily identified in the source code, and as such can be addressed. From discussions with developers of dozens of industrial systems we learn that the metrics are well accepted, or acceptable. The metrics are:

Total size

It is fairly intuitive that the total size of a system should feature heavily in any measure of maintainability. A larger system requires, in general, a larger effort to maintain. We use a simple line of code metric, which counts all lines of source code that are not comment or blank lines. We do not try to correct for "expressivity" of programming languages, or try to estimate the amount of functionality in a system. This way, a system in Java is smaller than a system with the same functionality in Cobol because it requires less lines of code to type in.

Number of modules

A software system is decomposed into modules. The number of modules, and the ratio between the number of modules and the total lines of code is a measure of how "well" it is decomposed. Obviously, this measure can be refined through measurements such as coupling and cohesion, but as an initial estimate it turns out to be rather useful.

Most programming languages have a natural concept of a module: for Java and C# it can be a class or a compilation unit, for Cobol it is a program, for C it would be a file. We currently follow an informal definition, which should be formalized at a later stage.

Number of units

A module in a software system can be decomposed into "units". The unit is the smallest piece of code that can be executed individually. For some languages the notion of module and the notion of unit are the same. In Cobol, there is no smaller unit than a program. Further decompositions such as sections or paragraphs are effectively labels, but are not encapsulated pieces of code. The unit of code is the fragment of the system that can be "unit-tested" in the sense of automated unit testing frameworks such as JUnit (for Java) or NUnit (for .net).

In Java or C# a unit is a method, in C a unit is a procedure.

Cyclomatic Complexity above X

Since the unit is the smallest piece of a system that can be executed individually, it makes sense to calculate the cyclomatic complexity on that unit. As we discussed earlier, the complexity follows a power law distribution, so calculating an average will give a result that may smooth out the outliers. The summation of the individual complexities of each unit does not lead to a meaningful number, so another way to aggregate the complexity needs to be found.

To alleviate the problem we express complexity in the number of lines of code of the system that are in units that have a higher complexity than X.

If we take X to be 20, and a system has ten units, of which one has a higher complexity than 20, the number of lines of code, relative to the total lines of code of the system, will be reported.

Duplication

We have analyzed a number of systems that were larger (in lines of code) than we had intuitively expected. Although a clear argument can be made for the size of a system being a large factor in the maintenance effort, it immediately poses the question: "But how large should this system be, given this functionality?" We have found that measuring code duplication gives a fairly simple estimate of how much larger a system is than it needs to be. Of course, various other factors contribute to a system being larger than necessary, including the lack of use of library functions.

Calculating straightforward code duplication has the advantage of giving an indication, without having to know anything about the functionality of the system.

We calculate code duplication as the percentage of all code that occurs more than once in equal code blocks of at least X lines. So if a single statement is repeated many times, but the statements before and after differ every time, we do not count it as duplicated. If however, a group of X statements (lines) appears unchanged in more than one place, we count it as duplicated. In general, we take X to be 6, which seems to give relevant results. The duplication we measure is an exact string matching duplication. We do not look at program dependency graphs [6], or other ways to determine potential duplication, but merely at exact string matching. This makes the process relatively easy, and not much less useful.

Discussion

It would be nice to have target values for each element of our maintainability model, to know when a system is maximally maintainable. Of course, this is impossible, not in the least because it is impossible to determine how many lines of code a system should have. However, some targets can be set. We offer our untested hypothesis:

- Lines of code should be as small as possible: a system that does the same with fewer lines of code is more maintainable
- The number of modules should be as small as possible
- The number of units should be as small as possible
- The percentage of lines of code that are in a unit that has a cyclomatic complexity of more than X should be zero. We currently take X to be twenty, although research (McCabe) suggests that X should be 10 [7].
- The percentage of duplicated code should be zero, depending on the block size of the duplicated blocks. If we take the block size to be 6 lines, we see a duplication percentage of around 3% for well managed systems.

The proposed model does not suffer from the problems identified for the Maintainability Index. It does not generate a single number, so it is not a composite index. It facilitates root cause analysis better than the MI, because it does not use averages. It can be easily explained to both technical personnel as well as to responsible managers. It uses numbers that can be easily influenced by changing the code. Preliminary findings show that these changes in the code make systems more maintainable, according to the maintainers of the systems.

Conclusion and Future Work

We have been using this model for a number of months now, and have applied it to a dozen different software systems. So far, it appears that the various components of the model balance each other. The duplication percentage directly offers a rationalization for the number of lines of code. The maximum complexity cap limits the size of units. This in turn balances the “as few units as possible” rule, which, if unchecked, would lead to a single unit for the whole system (which clearly is not very maintainable). There is no direct counterweight for the “as few modules as possible” rule, which may lead the reader to think that a system with a single module is the most maintainable system. Obviously, this is not the case. Currently no one is developing systems with this model as a guide, so there is no problem there. However, this is a clearly a weak point in the model, which needs to be addressed.

We are currently putting this model to work in our consultancy practice. So far, about a dozen systems have been measured using this model. Since the results are preliminary, we cannot yet report them, but they appear encouraging. Based on interviews with developers and maintainers of the systems that were analyzed, our initial finding is that the model seems to be significant with respect to the intuition of the developers, and in most case with the actual amount of time (and money) spent on maintenance.

We are setting up a benchmark database to study data for more systems, and ask interested readers who have access to software systems for which they can calculate the numbers mentioned in this paper to contribute their measurement data.

References

[1] Oman, P. & Hagemeister, J. "Metrics for Assessing a Software System's Maintainability," 337-344. *Conference on Software Maintenance 1992*. Orlando, FL, November 9-12, 1992. Los Alamitos, CA: IEEE Computer Society Press, 1992.

[2] Halstead, Maurice H. *Elements of Software Science*, Operating, and Programming Systems Series Volume 7. New York, NY: Elsevier, 1977.

[3] McCabe, Thomas J., "A Complexity Measure," *IEEE Transactions on Software Engineering,* SE-2 No. 4, pp. 308-320, December 1976.

[4] Szulewski, Paul, et al. *Automating Software Design Metrics* (RADC-TR-84-27). Rome, NY: Rome Air Development Center, 1984.

[5] Jones, Capers. "Software Metrics: Good, Bad, and Missing." Computer 27, 9 (September 1994): 98-100.

[6] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. 1998. Clone Detection Using Abstract Syntax Trees. In Proceedings of the international Conference on Software Maintenance (March 16 - 19, 1998). ICSM. IEEE Computer Society, Washington, DC, 368.

[7] McCabe, Thomas J. & Watson, Arthur H. "Software Complexity." Crosstalk, Journal of Defense Software Engineering 7, 12 (December 1994): 5-9.