The Software Maintainability Index Revisited

Kurt D. Welker
Idaho National Engineering and Environmental Laboratory

In 1991 Oman and Hagemeister introduced a composite metric for quantifying software maintainability. This Maintainability Index (MI) has evolved into numerous variants and has been successfully applied to a number of industrial strength software systems. After nearly a decade of use, MI continues to provide valuable insight into software maintainability issues. This article presents some of the author's observations about the practical use of MI in determining software maintainability.

For many years now, software practitioners have been collecting metrics from source code in an effort to better understand the software they are developing or changing. Maintainability Index (MI) is a composite metric that incorporates a number of traditional source code metrics into a single number that indicates relative maintainability. As originally proposed by Oman and Hagemeister, the MI is comprised of weighted Halstead metrics (effort or volume), McCabe's Cyclomatic Complexity, lines of code (LOC), and number of comments [1, 2]. Two equations were presented: one that considered comments and one that did not.

The original polynimial equations defining MI are as follows:

3-Metric: $MI=171-3.42\ln(aveE)-0.23aveV(g')-16.2\ln(aveLOC)$

where *aveE* is the average Halstead Effort per module, *aveV(g')* is the average extended cyclomatic complexity per module, and *aveLOC* is the average lines of code per module.

4-Metric: $MI=171-3.42\ln(aveE)-0.23aveV(g')-16.2\ln(aveLOC)+0.99aveCM$

where *aveE* is the average Halstead Effort per module, *aveV(g')* is the average extended cyclomatic complexity per module, *aveLOC* is the average lines of code per module, and *aveCM* is the average number of lines of comments per module.

The rationale behind this selection of metrics was to construct a rough order, composite metric that incorporated quantifiable measurements for the following:

- Density of operators and operands (how many variables and how they are used).
- Logic complexity (how many execution paths are in the code).
- Size (how much code is there).
- Human insight (comments in the code).

Other variants of the MI have evolved using slightly different metrics, metric combinations, and weights [3, 4, 5]. Each has the general flavor of the basic MI equation and underlying rationale.

> *"Subjective measures applied via human code reviews still play an extremely important role in assessing software maintainability."*

Reasonable success has been achieved in using MI to quantify and improve software maintainability both during development and maintenance activities [4, 6, 7, 8, 9, 10, 11,].

Practically speaking though, the MI is only one piece in understanding the maintainability puzzle. Furthermore, it should not be interpreted in a vacuum. Rather it should be used as an indicator to direct human investigation and review. The following sections discuss some practical insight in applying MI to typical software systems.

Discussion of MI Equations

Several variants of the MI equations have evolved over time. Some academic opinion places more confidence in Halstead's Volume Metric than his Effort Metric so the MI equations were adjusted to incorporate the use of Halstead's Volume. Additionally, studies have shown that the MI model was often overly sensitive to the comment metric in the 4-Metric equation and thus that portion of the equation was modified to limit the contribution of comments in MI [11, 12].

The typical modified MI equations look similar to the following:

3-Metric: $MI = 171 -5.2\ln(aveV) -0.23aveV(g') - 16.2\ln(aveLOC)$

where *aveV* is the average Halstead Volume per module, *aveV(g')* is the average extended cyclomatic complexity per module, and *aveLOC* is the average lines of code per module.

4-Metric: $MI = 171 -5.2\ln(aveV) -0.23aveV(g') -16.2\ln(aveLOC) +50.0\sin\sqrt{2.46\ perCM}$

where *aveV* is the average Halstead Volume per module, *aveV(g')* is the average extended cyclomatic complexity per module, *aveLOC* is the average lines of code per module, and *perCM* is the average percent of lines of comments per module.

Examination of these equations indicates that picking the appropriate MI equation is still a subject for discussion[1]. The consideration of comments in the MI is a big discussion point. First, if a human assessment of the software concludes that the majority of the comments in the software are correct and appropriate, then the 4-Metric MI is potentially appropriate. Otherwise, the 3-Metric equation is probably a better fit. Second, if the 4-Metric equation is selected, it is still possible that the comments may inappropriately skew the MI. New research has been performed and additional modifications have been proposed to further refine the MI [13]. These refinements appear to add stability to the behavior of the MI for assessing specific types of software systems.

Observations

As mentioned earlier, numerous papers have been written describing the successful application of MI as part of the software development process or within a software

maintenance assessment. What is often not discussed are some simple, common sense guidelines that should be considered when using an objective metric such as MI. Presented in the next few paragraphs are some general observations the author has gathered from applying MI to a variety of software systems.

Comments in the Code: Comments in the source code are a two-edged sword when it comes to considering their role in software maintenance. Accurate, up-to-date comments that provide additional insight not already obvious from the source code are generally quite helpful when it comes to making changes later on. However, comments that have not continued to evolve with their associated software can actually be a maintenance hindrance.

Comments, just like source code, will degrade over time as maintenance activities are performed unless specific actions are taken to keep them from becoming inaccurate. Comments are not executed at run-time; they are usually for people. Only people can tell if the comments in the code are helpful or not. Just because there are comments in the code does not mean that the code is more maintainable.

Furthermore, not all real comments are detectable by automated tools. For instance, when writing self-documenting code, some software developers put engineering units in the variable names (distanceFt or effectivePowerW). These types of comments are quite helpful to the human developer, ignored by the compiler and also most metrics extraction tools, yet certainly make the software more maintainable.

When determining how an automated tool will credit maintainability for comments in the code, a human must first determine the quality and usefulness of the comments. A variant MI equation could be developed that penalizes maintainability based on the poor quality of the comments. Current 4-Metric MI equations that include a metric for comments, must be applied with some human insight. Comparisons between the 3-Metric MI and the 4-Metric MI are also helpful in flagging source code with inappropriate comments. One guideline is that when there is more than a 15-point delta between the 3-Metric MI and the 4-Metric MI, comments in the associated source code should be manually examined for appropriateness.

Here is the bottom line on MI and comments: A man in the maintainability assessment loop is essential both in deciding how to measure comments in the source code (which MI equations) and then in determining the meaning of the results (do the comments make the software easier to maintain).

"Comments in the source code are a two-edged sword ..."

Interpretation of Results: Source code metrics provide only an objective measure. MI works the same. Subjective measures applied via human code reviews still play an extremely important role in assessing software maintainability. After all people maintain the software. Automated maintenance is not a reality yet. Therefore, it only makes sense that there are some characteristics of software construction that take a person to quantify for attributes such as maintainability. Determining maintainability purely by objective measures can be deceiving. Take for instance the following simplistic example. Consider two versions of a program that print the words to "The Twelve Days of Christmas" (see Figure 1, page 20). Sorry but I don't remember whom to credit for writing Example 1. A few metrics for the two versions are as follows in Figure 2, page 20.

Possible interpretations might include: Based on the 3-Metric MI alone, Example 1 is slightly more maintainable; based on cyclomatic complexity alone, Example 1 is more maintainable; based on lines of code, Example 1 is more maintainable, i.e. less code to maintain? Based on the 4-Metric MI, Example 2 is more maintainable, but did the comments really make the difference? No, the comments are okay but they are not the driving factor for judging maintainability. Based on effort alone, Example 2 is more maintainable than Example 1.

What made your decision regarding which source was more maintainable? It was probably the human examination of the source code. Which version of the source would you want to maintain? Especially when you learn that Example 1 contains a bug as the word eighth is misspelled in the output. This example may be extreme, but it illustrates the point. The metrics for real-world software can present similar difficulties.

Object Oriented Decomposition and Fracturing: Software languages, architectures, and decomposition techniques have evolved since the original development and validation of the MI equations. Object-oriented analysis and design have influenced the structure of today's software systems. Typical object-oriented software tends to be decomposed into smaller modules than software systems that were decomposed using other techniques. Consider all the get and set methods in a typical object class. What impact does having a large number of smaller modules play in the MI?

Object-oriented software is fundamentally composed of operators and operands and has a number of executable paths through the code. The lines of code may still be counted and commented. From this perspective, MI still provides a good fit [11]. But additionally, there are constructs such as classes and inheritance that could be considered in tailoring MI for an object-oriented system.

Discussion of these types of MI enhancements will be postponed for another time. It appears, though, that object-oriented systems by nature have a fairly high MI due to the typical smaller module size. Naturally, smaller modules contain less operators and operands, less executable paths, and less lines of comments and code; therefore, the MI tends to be higher. It is the author's opinion that even so, the MI is still applicable for object-oriented systems, but that maybe the maintainability classification thresholds should be raised when interpreting MI's from object-oriented systems. MI is still great for identifying overly complex (and therefore difficult to maintain) modules in object-oriented systems.

Is it possible to decompose a software system into modules that are too small? Yes, software fracturing can occur and when that happens, modules loose cohesion and the coupling between modules

Example 1

```
#include <stdio.h>
main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')){}#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
 :t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
  :0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);
}
```

Example 2

```
/****************************
**
** Print the "Twelve Days of Christmnas"
**
****************************/
#include <stdio.h>
main()
{
int i;
char *int_to_word();

  for (i=1; i<=12; i++) {

    printf ("On the %s day of Christmas my true love gave to me\n",
          int_to_word(i));

    switch (i) {
      case 12: printf("twelve drummers drumming, ");
      case 11: printf("eleven pipers piping, ");
      case 10: printf("ten lords a-leaping,\n");
      case 9: printf("nine ladies dancing, ");
      case 8: printf("eight maids a-milking, ");
      case 7: printf("seven swans a-swimming,\n");
      case 6: printf("six geese a-laying, ");
      case 5: printf("five gold rings;\n");
      case 4: printf("four calling birds, ");
      case 3: printf("three french hens, ");
      case 2: printf("two turtle doves\nand ");
      case 1: printf("a partridge in a pear tree.\n\n");
        }
  }
}

/****************************
**
** Convert the specified integer to English text
**
****************************/
char *int_to_word (i)
int i;
{
  switch (i) {
        case 1: return ("first");
        case 2: return ("second");
        case 3: return ("third");
        case 4: return ("fourth");
        case 5: return ("fifth");
        case 6: return ("sixth");
        case 7: return ("seventh");
        case 8: return ("eighth");
        case 9: return ("ninth");
        case 10: return ("tenth");
        case 11: return ("eleventh");
        case 12: return ("twelfth");
  }
}
```

Figure 1: *The Twelve Days of Christmas - Sample Code*

increases. When fracturing occurs, the maintainability of the system (from a human perspective) actually decreases, and metrics such as MI are not necessarily a realistic measure of the actual maintainability. Controlled software development processes, good software engineering practices, and code reviews again become key in assuring and assessing maintainability. Thus it is evident that high MI does not guarantee the code is maintainable. A man in the loop is still essential.

Figure 2: *The Twelve Days of Christmas - Metrics*

	EXAMPLE 1	EXAMPLE 2
3 metric MI	87.4	86.7
4 metric MI	87.4	117.9
Effort	25668	6840
V (g')	13	27
LOC	17	54
Comments	0	10

Conclusion

Using the MI to assess source code and thereby identify and quantify maintainability is an effective approach. The MI provides one small perspective into the highly complex issues of software maintenance. The MI provides an excellent guide to direct human investigation. Hopefully, this paper provides some insight to the practical use of MI. To recap, continue to comment source code but do not put too much faith in comments to improve maintainability. Continue to measure maintainability using MI but do not interpret the results in a vacuum. Be aware of the limitations of objective metrics such as MI. Changing technologies will require changing metrics.◆

References

1. Oman, P.W.; Hagemeister, J.; and Ash, D., A Definition and Taxonomy for Software Maintainability, Technical Report #91-08-TR, Software Engineering Test Laboratory, University of Idaho, Moscow, ID, 1991.
2. Oman, P.W. and Hagemeister, J., (1992) Metrics for Assessing a Software System's Maintainability, Proceedings of the Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 337-344.
3. Coleman, D., Assessing Maintainability, Proceedings of the Software Engineering Productivity Conference 1992, Hewlett-Packard, Palo Alto, CA, 1992, pp. 525-532.
4. Coleman, D.; Ash, D.; Lowther, B.; and Oman, P.W., Using Metrics to Evaluate Software System Maintainability, *IEEE Computer*, 1994, 27(8), pp. 44-49.
5. Oman, P.W. and Hagemeister, J., Constructing and Testing of Polynomials Predicting Software Maintainability, *Journal of Systems and Software*, 1994, 24(3), pp. 251-266.
6. Ash, D.; Alderete, J.; Yao, L.; Oman, P.W.; and Lowther, B., Using Software Maintainability Models to Track Code Health, Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA, 1994, pp. 154-160.
7. Coleman, D.; Lowther, B.; and Oman, P.W., The Application of Software Maintainability Models on Industrial Software Systems, *Journal of Systems and Software*, 1995, 29(1), pp. 3-16.
8. Oman, P.W., Applications of an Automated Source Code Maintainability Index, Technical Report #95-08-SL, Software Engineering Test Laboratory, University of Idaho, Moscow, ID, presented at the 1995 Software Technology Conference, Salt Lake City, UT.
9. Pearse, T. and Oman, P.W., Maintain-

ability Measurements on Industrial Source Code Maintenance Activities, Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA, 1995, pp. 295-303.
10. Welker, K. and Oman, P.W., Software Maintainability Metrics Models in Practice, CROSSTALK, Nov./Dec. 1995, pp. 19-23 and 32.
11. Welker, K.; Oman, P.W.; and Atkinson, G., Development and Application of an Automated Source Code Maintainability Index, *Journal of Software Maintenance*, 1997, May/June, pp. 127-159.
12. Lowther, B., The Application of Software Maintainability Metric Models on Industrial Software Systems, master's thesis, Department of Computer Science, University of Idaho, Moscow, ID, 1993.
13. Liso, A., Software Maintainability Metrics Model: An Improvement in the Coleman-Oman Model," CROSSTALK, Aug. 2001, pp. 15-17.

Note

1. The discussions in this article can apply to either set of MI definitions. The majority of people use the latter set of MI definitions. I still use the original MI equations for some applications. If used to track software over its life, it is important not to change equations mid-stream. There are other variants of the MI equations that organizations have tailored for specific interests (both the 3- and 4-metric versions). The discussion in the paper generally applies to most of these as well.

About the Author

Kurt D. Welker is an advisory engineer at the Idaho National Engineering and Environmental Laboratory with 14 years experience in software development, systems integration, and software measurement. He is a technical lead on the Electronic Combat System Integration Project performing reengineering, integration, and software maintenance on several electronic combat analysis models for the Air Force Information Warfare Center that simulate radar detection, weapon lethality envelopes, electronic counter-measures, reconnaissance, passive detection, and communications jamming. He functioned as the principle investigator for the development of a general-purpose lexical scanner/parser tool called the Data Stream Analyzer that provides data format integration. He also functioned as the principle investigator on a software measurement/process-improvement research initiative. He has been using MI to assess and track software maintainability for about eight years. Welker has a bachelor's of science degree in computer science from Brigham Young University and a master's of science degree in computer science from the University of Idaho.

Idaho National Engineering
and Environmental Laboratory
Idaho Falls, Idaho
E-mail: wdk@inel.gov

Letter to the Editor

Dear CROSSTALK,

I was reading the new June 2001 issue Vol. 14 No. 6 yesterday and was nonplussed to read in three different places (From the Publisher, the abstract to the first article *Extending UML to Enable the Definition and Design of Real-Time Embedded Systems*, and the text of *The Quality of Requirements in Extreme Programming*), references to Universal Markup Language (UML).

All three of the contexts refer to the Unified Modeling Language created by Booch, Rumbaugh, and Jacobson of Rational Software Corporation. There is no real-time software design methodology called Universal Markup Language to my knowledge.

Thanks for an excellent publication.

Regards,
Karl Woelfer
Seattle, WA

Coming Events

August 27-30
Software Test Automation Conference
www.sqe.com/testautomation

August 27-31
5th IEEE International Symposium on Requirements Engineering
www.re01.org

Sept. 10-14
Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering
www.esec.ocg.at

Oct. 15-18
16th Annual SEI Symposium
www.asq.org/ed/conferences

Oct. 15-19
21st International Conference on Software Testing and EXPO 2001
www.qaiusa.com/conferences

Oct. 22-24
11th International Conference On Software Quality
www.asq.org/ed/conferences

Oct. 29-Nov. 2
Software Testing Analysis and Review
www.sqe.com/starwest

Nov. 4-7
Amplifying Your Effectiveness (AYE)
www.ayeconference.com

Feb. 4-6, 2002
International Conference on COTS-Based Software Systems (ICCBSS) At the Heart of the Revolution
www.iccbss.org

April 28 - May 3, 2002
STC 2002
"Forging the Future of Defense Through Technology"
www.stc-online.org