COMPUTING PRACTICES

Using Metrics to Evaluate Software System Maintainability

In this month's Computing Practices we offer a sneak preview of *Computer's* September issue on software metrics. Software metrics have been much criticized in the last few years, sometimes justly but more often unjustly, because critics misunderstand the intent behind the technology. Software complexity metrics, for example, rarely measure the "inherent complexity" embedded in software systems, but they do a very good job of comparing the relative complexity of one portion of a system with another. In essence, they are good modeling tools. Whether they are also good measuring tools depends on how consistently and appropriately they are applied. The two articles showcased here suggest ways of applying such metrics.

Our first article, by Don Coleman et al., sets forth maintainability metrics for gauging the effect of maintenance changes in software systems, rank ordering subsystem complexity, and comparing the "quality" of two different systems.

The second article, by Norman Schneidewind, describes an approach to validating software quality metrics for large-scale projects such as the space shuttle flight software. The proposed metrics isolate specific quality factors that let us predict and control software quality.

Please feel free to contact me directly about articles you liked, didn't like, or would like to see in this section (oman@cs.uidaho.edu).

—*Paul Oman*

Don Coleman and Dan Ash, Hewlett-Packard

Bruce Lowther, Micron Semiconductor

Paul Oman, University of Idaho

With the maturation of software development practices, software maintainability has become one of the most important concerns of the software industry. In his classic book on software engineering, Fred Brooks[1] claimed, "The total cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it." Parikh[2] had a more pessimistic view, claiming that 45 to 60 percent is spent on maintenance. More recently, two recognized experts, Corbi[3] and Yourdon,[4] claimed that software maintainability is one of the major challenges for the 1990s.

These statements were validated recently by Dean Morton, executive vice president and chief operating officer of Hewlett-Packard, who gave the keynote address at the 1992 Hewlett-Packard Software Engineering Productivity Conference. Morton stated that Hewlett-Packard (HP) currently has between 40 and 50 million lines of code under maintenance and that 60 to 80 percent of research and development personnel are involved in maintenance activities. He went on to say that 40 to 60 percent of the cost of production is now maintenance expense.

The intent of this article is to demonstrate how automated software maintainability analysis can be used to guide software-related decision making. We have applied metrics-based software maintainability models to 11 industrial software systems and used the results for fact-finding and process-selection decisions. The results indicate that automated maintainability assessment can be used to support buy-versus-build decisions, pre- and post-reengineering analysis, subcomponent quality analysis, test resource allocation, and the prediction and targeting of defect-prone subcomponents. Further, the analyses can be conducted at various levels of granularity. At the component level, we can use these models to monitor changes to the system as they occur and to predict fault-prone components. At the file level, we can use them to identify subsystems that are not well organized and should be targeted for perfective maintenance. The results can also be used to determine when a system should be reengineered. Finally, we can use these models to compare whole systems. Comparing a known-quality system to a third-party system can provide a basis for deciding whether to purchase the third-party system or develop a similar system internally.

0018-9162/94/$4.00 © 1994 IEEE

Recent studies in metrics for software maintainability and quality assessment have demonstrated that the software's characteristics, history, and associated environment(s) are all useful in measuring the quality and maintainability of that software.[5-7] Hence, measurement of these characteristics can be incorporated into software maintainability assessment models, which can then be applied to evaluate industrial software systems. Successful models should identify and measure what most practitioners view as important components of software maintainability.

A comparison of five models

We recently analyzed five methods for quantifying software maintainability from software metrics. The definition, derivation, and validation of these five methods has been documented elsewhere.[7] Only a synopsis of the five methods is presented here:

- *Hierarchical multidimensional assessment models* view software maintainability as a hierarchical structure of the source code's attributes.[6]
- *Polynomial regression models* use regression analysis as a tool to explore the relationship between software maintainability and software metrics.[8]
- *An aggregate complexity measure* gauges software maintainability as a function of entropy.[5]
- *Principal components analysis* is a statistical technique to reduce collinearity between commonly used complexity metrics in order to identify and reduce the number of components used to construct regression models.[7]
- *Factor analysis* is another statistical technique wherein metrics are orthogonalized into unobservable underlying factors, which are then used to model system maintainability.[5]

Tests of the models indicate that all five compute reasonably accurate maintainability scores from calculations based on simple (existing) metrics. All five models and the validation data were presented to HP Corporate Engineering managers in the spring and summer of 1993. At that time it was decided that the hierarchical multidimensional assessment and the polynomial regression models would be pursued as simple mechanisms for maintainability assessment that could be used by maintenance engineers in a variety of locations. HP wanted quick, easy-to-calculate indices that "line" engineers could use at their desks. The following subsections explain how these methods were applied to industrial systems.

HPMAS: A hierarchical multidimensional assessment model. HPMAS is HP's software maintainability assessment system based on a hierarchical organization of a set of software metrics. For this particular type of maintainability problem, Oman and Hagemeister[6] have suggested a hierarchical model dividing maintainability into three underlying dimensions or attributes:

(1) The *control structure*, which includes characteristics pertaining to the way the program or system is decomposed into algorithms.

(2) The *information structure*, which includes characteristics pertaining to the choice and use of data structure and dataflow techniques.

(3) *Typography, naming, and commenting*, which includes characteristics pertaining to the typographic layout, and naming and commenting of code.

We can easily define or identify separate metrics that can measure each dimension's characteristics. Once the metrics have been defined and/or identified, an "index of maintainability" for each dimension can be defined as a function of those metrics. Finally, the three dimension scores can be combined for a total maintainability index for the system. For our work, we used existing metrics to calculate a deviation from acceptable ranges and then used the inverse of that deviation as an index of quality.

Most metrics have an optimum range of values within which the software is more easily maintained. A method called *weight and trigger-point-range analysis* is

Software maintenance definitions

Definitions for software "maintenance" and "maintainability" are many, but they are fairly consistent in scope and intent. Here are the IEEE standard definitions:[1]

Maintenance: The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.

Maintainability: The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.

Consistent with these definitions, the maintenance process can be divided into three areas of focus:[1]

Corrective maintenance: Maintenance performed to correct faults in hardware or software.

Adaptive maintenance: Software maintenance performed to make a computer program usable in a changed environment.

Perfective maintenance: Software maintenance performed to improve the performance, maintainability, or other attributes of a computer program.

Reference

1. IEEE Std. 610.12-1990, "Glossary of Software Engineering Terminology," in *Software Engineering Standards Collection*, IEEE CS Press, Los Alamitos, Calif., Order No. 1048-06T, 1993.

used to quantify maintainability by calculating a "degree of fit" from a table of acceptable metric ranges. When the metric value falls outside the optimum range, it indicates that maintainability is lower; hence, there is a deviation (or penalty) on the component's contribution to maintainability. The optimum range value, called the *trigger point range*, reflects the "goodness" of the program style. For example, if the acceptable range for *average lines of code* (aveLOC) is between 5 and 75, values falling below 5 and above 75 serve as the trigger points for what would be classified as poor style. If the measured average lines of code value lies within the acceptable range, there is no penalty. If the metric value falls outside the trigger point range but is close to the bounds (trigger points), we then apply a proportional deviation, which can run up to 100 percent (the maximum penalty). The weighted deviation is computed by multiplying the calculated deviation by a weighted value between zero and one, inclusive. The metric attributes are combined based on the assumption that the dimensional maintainability is 100 percent (highly maintainable); they are then reduced by the deviation percentage of each metric. Dimension maintainability is calculated as

$$DM_{\text{dimension}} = 1 - \frac{\sum w_i D_i}{\sum w_i}$$

The overall maintainability index is the product of the three dimensions. Multiplying the three dimensions' maintainability gives a lower overall maintainability than averaging does, which underscores the fact that deviation in one aspect of maintainability will hinder other aspects of the maintenance effort, thus reducing maintainability of the entire system.

HPMAS was calibrated against HP engineers' subjective evaluation of 16 software systems, as measured by an abridged version of the AFOTEC (Air Force Operational Test and Evaluation Center) software quality assessment instrument.[9] HPMAS maintainability indices range from 0 to 100, with 100 representing excellent maintainability.

Polynomial assessment tools. Regression analysis is a statistical method for predicting values of one or more response (dependent) variables from a collection of predictor (independent) variables. For purposes of software maintainability assessment, we need to create a polynomial equation by which a system's maintainability is expressed as a function of the associated metric attributes. We have used this technique to develop a set of polynomial maintainability assessment models.[8] These models were developed as simple software maintainability assessment methods that could be calculated from existing metrics. Since these models were intended for use by maintenance practitioners "in the trenches," the models were again calibrated to HP engineers' subjective evaluation of the software as measured by the abridged version of the AFOTEC software quality assessment instrument.[9] That is, the independent variables used in our models were a host of 40 complexity metrics, and the dependent variable was the (numeric) result of the abridged AFOTEC survey.

Approximately 50 regression models were constructed in an attempt to identify simple models that could be calculated from existing tools and still be generic enough to apply to a wide range of software systems. In spite of the current research trend away from the use of Halstead metrics, all tests clearly indicated that Halstead's volume and effort metrics were the best predictors of maintainability for the HP test data. The regression model that seemed most applicable was a four-metric polynomial based on Halstead's effort metric and on metrics measuring extended cyclomatic complexity, lines of code, and number of comments:

$$\begin{aligned}\text{Maintainability} &= 171 \\ &- 3.42 \times \ln(aveE) \\ &- 0.23 \times aveV(g') \\ &- 16.2 \times \ln(aveLOC) + aveCM\end{aligned}$$

where *aveE*, *aveV(g')*, *aveLOC*, and *aveCM* are the average effort, extended V(G), average lines of code, and number of comments per submodule (function or procedure) in the software system.

Preliminary results indicated that this model was too sensitive to large numbers of comments. That is, large comment blocks, especially in small modules, unduly inflated the resulting maintainability indices. To rectify this, we replaced the *aveCM* component with percent comments (*perCM*), and a ceiling function was placed on the factor to limit its contribution to a maximum value of 50.[10] Also, because there has been much discussion of the nonmonotonicity of Halstead's effort metric (it is not a nondecreasing function under the concatenation operation), we reconstructed the model using Halstead's volume metric instead. Thus, the final four-metric polynomial now used in our work is

$$\begin{aligned}\text{Maintainability} &= 171 \\ &- 5.2 \times \ln(aveVol) \\ &- 0.23 \times ave\ V(g') \\ &- 16.2 \times \ln(aveLOC) \\ &+ (50 \times \sin(\sqrt{2.46 \times perCM}))\end{aligned}$$

This polynomial has been compared to the original model using the same validation data. The average residual between the effort-based model and the volume-based model is less than 1.4.

Applying the models to industrial software

A software maintainability model is only useful if it can provide developers and maintainers in an industrial setting with more information about the system. Hence, the data used to test and validate our models consisted entirely of genuine industrial systems provided by Hewlett-Packard and Defense Department contractors. The examples are presented here to show how these models can aid software maintainers in their decision making. The data presented in the following subsections is real and unaltered, except that proprietary information has been removed.

Using HPMAS in a pre/postanalysis of maintenance changes. Over several years of software maintenance, systems tend to

degrade as the number of "patches" to them increases. To combat this increase in entropy, a pre/postanalysis can be used to ensure that the maintainability of a system does not decline after each maintenance modification. To exemplify this, an existing HP subsystem, written in C for the Unix platform, was analyzed using HPMAS prior to perfective maintenance modification. Once the modification was complete, the modified subsystem was analyzed by HPMAS and the results were compared to determine if there was any detectable change in the maintainability of the subsystem. Table 1 contains an overall analysis of the changes made to the subsystem.

The HPMAS maintainability index in Table 1 shows that the maintainability of the subsystem was essentially unchanged (a 0.4 percent increase) even though the perfective maintenance changes had actually increased the complexity of the system. Specifically, 149 lines of code, two modules, and 29 branches were added to the system. Although the maintenance engineer denied that functionality increased, a visual inspection of the source code revealed that increased error checking had, in fact, been added to the code. For example, the original version of module Function_F, shown in section 2 of Table 2, contained 12 error-screening checks, while the modified version contained 16 error checks. (Throughout this discussion, function names have been changed to protect Hewlett-Packard proprietary information.)

Table 2 contains a module-by-module comparison of the pre- and post-test maintainability indices for the subsystem. The table is divided into four sections to demonstrate the distribution of maintenance changes. The first section of the table contains the modules that were not modified during the maintenance task. The second section contains modules that were slightly modified but which retained their original module names. The third section contains modules that have been modified and renamed. (The modules in this section were matched by visually inspecting the post-test system to identify any reused comments, variables, or control flow used in the pretest system.) The last section contains modules in the pretest system that could not be matched to any module in the post-test system. (Visual inspection of the code revealed that the post-test components contained reused code from the pretest system, but they could not be matched to any one post-test component.) Thus, the last section represents an area of the program where the subsystem was repartitioned, resulting in a new subsystem organization.

This type of postmaintenance analysis can provide the maintenance staff with a wealth of information about the target system. For example, section 1 of Table 2 consists of unchanged components with relatively high HPMAS maintainability scores. If these components remain unchanged over several maintenance modifications, they might be considered for a reusability library. Components in the second section address the system goal but have not yet reached the refinement of those in the first section. Their HPMAS metrics are generally lower than those in the first section, and they have changed less than ± 5 percent from the pre- to postanalysis.

Table 1. Comparing pre- and post-test results shows how much maintenance modification changes a subsystem.

	Pretest	Post-test	Percent Change
Lines of code	1,086.00	1,235.00	13.4
Number of modules	13.00	15.00	15.4
Total V(g′)	226.00	255.00	12.8
HPMAS maintainability index	88.17	88.61	0.4

Table 2. Module-by-module comparison of pre- and postanalysis results.

Section	Pretest Analysis		Post-test Analysis		Percent
	Name	Metric	Name	Metric	Change
1	Function_A	93.83	Function_A	93.83	0.0
	Function_B	93.82	Function_B	93.82	0.0
	Function_C	92.96	Function_C	92.96	0.0
	Function_D	84.41	Function_D	84.41	0.0
2	Function_E	86.24	Function_E	89.00	3.2
	Function_F	65.58	Function_F	67.27	2.6
	Function_G	88.06	Function_G	85.83	–2.5
3	Function_H	78.41	Function_H′	83.05	5.9
	Function_I	72.85	Function_I′	63.15	–13.3
	Function_J	67.75	Function_J′	66.43	–1.9
	Function_K	68.83	Function_K′	66.67	–3.1
4	Function_L	80.68			
	Function_M	78.78			
			Function_N	85.08	
			Function_O	80.75	
			Function_P	79.68	
			Function_Q	69.68	

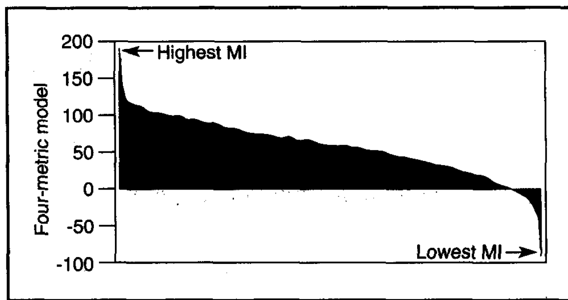

Figure 1. Maintainability index plot for 714 files that make up a software system with 236,000 lines of code.

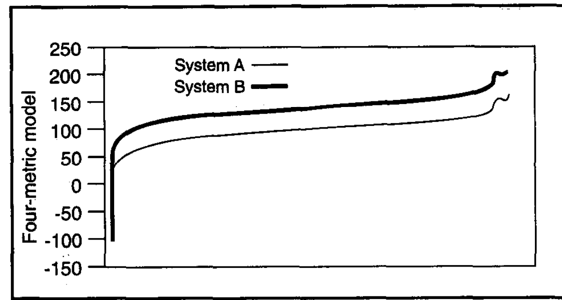

Figure 2. Using the four-metric model to compare maintainability of two software systems.

The last two sections contain the most extensive changes to the subsystem. Components in these two sections represent a large burden to the maintainer, essentially representing a repartition of the problem. This is evidenced by the renaming of components, lower HPMAS metric values, and unmatchable pre- and post-test components. The maintenance engineer renamed all of the components in section 3 (presumably because he thought the original names did not adequately describe them) and substantively changed their functionality. Section 4 contains old components that could not be matched to components in the new system. They represent the largest burden to the maintenance effort because (1) the new components are untested, (2) the structure of the system has changed, requiring all documentation and diagrams for this system to be updated, and (3) all maintainers who were familiar with the pretest system are unfamiliar with the post-test system.

Using polynomials to rank-order module maintainability. To detect differences in subsystem maintainability, the four-metric polynomial was applied to a large third-party software application sold to HP. The system consists of 236,000 lines of C source code written for a Unix platform. The software complexity metrics were calculated on a file-by-file basis, and a maintainability index was calculated for each file.

The file-by-file analysis of the 714 files constituting the software system is shown in Figure 1. This histogram shows the maintainability (polynomial) index for each file, ordered from highest to lowest. The index for each file is represented by the top of each vertical bar; for negative indices, the value is represented by the bottom of the bar. The maintainability analysis for this system showed that the file maintainability scores (or indices) range from a high of 183 to a low of – 91.

All components above the 85 maintainability index are highly maintainable, components between 85 and 65 are moderately maintainable, and components below 65 are "difficult to maintain." The dotted line indicates the *quality cutoff* established by Hewlett-Packard at index level 65.[10] Although these three quality categories are used by HP, they represent only a good "rule of thumb."

The figure shows that 364 files, or roughly 50 percent of the system, fall below the quality-cutoff index, strongly suggesting that this system is difficult to modify and maintain. Prior to our analysis, the HP maintenance engineers had stated that the system was *very* difficult to maintain and modify. Further analysis proved that change-prone and defect-prone subsystem components (files) could be targeted using the ranked order of the maintainability indices.

In a subsequent study, a similar analysis was conducted on another third-party subsystem and compared against a maintainability index profile for a proprietary HP system (an example is shown in the next subsection). Based on that comparison, HP decided to purchase the third-party software.

Using polynomials to compare software systems. The polynomial models can also be used to compare whole software systems. We analyzed two software systems that were similar in size, number of modules, platform, and language (see Table 3).

The first system, A, is a third-party acquisition that had been difficult to maintain. (Again, the names of the two systems have been changed to protect proprietary information.) The second system, B, had been cited in internal Hewlett-Packard documentation as an excellent example of state-of-the-art software development. The four-metric polynomial model was used to compare the two systems to see the differences in their maintainability profiles. HP maintenance engineers, already experienced with the systems, were asked to comment on the maintainability of each system.

The results of the polynomial model shown in Table 3 corroborate the engineers' informal evaluation of the two software systems. The A system yielded a maintainability index of 89; while clearly above our acceptability criteria, it is considerably lower than the 123 maintainability index calculated for system B. This corresponds to the mediocre evaluation A received from the Hewlett-Packard engineers and the high praise B received from the engineers working on that system. We performed a more

Table 3. A polynomial comparison of two systems corroborated an informal evaluation by engineers.

	A	B
HP evaluation	Low	High
Platform	Unix	Unix
Language	C	C
Total LOC	236,275	243,273
Number of modules	3,176	3,097
Overall maintainability index	89	123

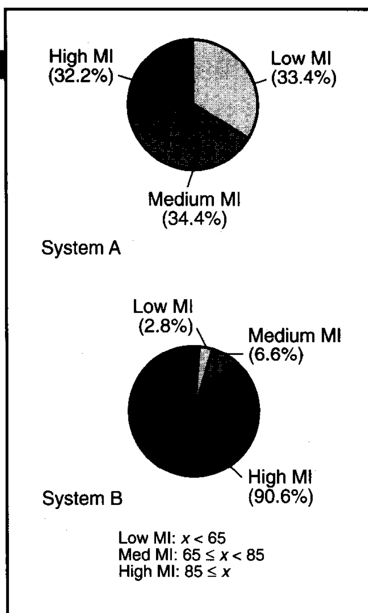

Figure 3. Comparison of two systems, with high-, medium-, and low-maintenance lines of code expressed in percentages.

granular analysis by calculating the polynomial on a module-by-module basis. Figure 2 shows a plot of the ordered results. The B system (the thick line) consistently scored higher than the A system for all but one module. The significant gap between the two plots accentuates the fact that the A system is less maintainable.

Figure 3 contains two pie charts showing the distribution of lines of code in the three maintainability classifications (high, medium, and low). The upper pie chart, representing the A system, illustrates the nearly equal distribution of code into the three classifications. The lower pie chart, representing the B system, shows that a significant portion of this system falls in the high maintainability classification. The B system contains only 15 components, representing 2.8 percent of the lines of code, that fall below the quality cutoff. The A system, on the other hand, contains 228 components, representing 33.4 percent of the lines of code, that fall below the quality cutoff. Hence, using lines of code to compare the two systems reveals that although their overall maintainability index is adequate, the B system is likely to be much easier to maintain than the A system. This result corresponds to the Hewlett-Packard evaluations.

To date we have conducted an automated software maintainability analysis on 11 software systems. In each case, the results from our analysis corresponded to the maintenance engineers' "intuition" about the maintainability of the (sub)system components. But in every case, the automated analysis provided additional data that was useful in supporting or providing credence for the experts' opinions.

Our analyses have assisted in buy-versus-build decisions, targeting subcomponents for perfective maintenance, controlling software quality and entropy over several versions of the same software, identifying change-prone subcomponents, and assessing the effects of reengineering efforts.

Software maintainability is going to be a considerable challenge for many years to come. The systems being maintained are becoming increasingly complex, and a growing proportion of software development staff is participating in the maintenance of industrial software systems. Our results indicate that automated maintainability analysis can be conducted at the component level, the subsystem level, and the whole system level to evaluate and compare software. By examining industrial systems at different levels, a wealth of information about a system's maintainability can be obtained. Although these models are not perfect, they demonstrate the utility of such models. The point is that a good model can help maintainers guide their efforts and provide them with much needed feedback. Before developers can claim that they are building maintainable systems, there must be some way to measure maintainability. ■

References

1. F.P. Brooks, *The Mythical Man–Month: Essays on Software Engineering*, Addison–Wesley, Reading, Mass., 1982.
2. G. Parikh and N. Zvegintzov, *Tutorial on Software Maintenance*, IEEE CS Press, Los Alamitos, Calif., Order No. 453, 1983.
3. T. Corbi, "Program Understanding: Challenge for the 1990s," *IBM Systems J.*, Vol. 28, No. 2, 1989, pp. 294-306.
4. E. Yourdon, *The Rise and Fall of the American Programmer*, Yourdon Press Computing Series, Trenton, N.J., 1992.
5. J. Munson and T. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Trans. Software Eng.*, Vol. 18, No. 5, May 1992, pp. 423-433.
6. P. Oman and J. Hagemeister, "Metrics for Assessing Software System Maintainability," *Proc. Conf. Software Maintenance,* IEEE CS Press, Los Alamitos, Calif., Order No. 2980-02T, 1992, pp. 337-344.
7. F. Zhuo et al., "Constructing and Testing Software Maintainability Assessment Models," *Proc. First Int'l Software Metrics Symp.*, IEEE CS Press, Los Alamitos, Calif., Order No. 3740-02T, 1993, pp. 61–70.
8. P. Oman and J. Hagemeister, "Construction and Testing of Polynomials Predicting Software Maintainability," *J. of Systems and Software*, Vol. 24, No. 3, Mar. 1994, pp. 251-266.
9. *Software Maintainability — Evaluation Guide*, AFOTEC Pamphlet 800-2 (updated), HQ Air Force Operational Test and Evaluation Center, Kirkland Air Force Base, N.M., Vol. 3, 1989.
10. D. Coleman, "Assessing Maintainability," *Proc. 1992 Software Eng. Productivity Conf.*, Hewlett-Packard, Palo Alto, Calif., 1992, pp. 525–532.

Don Coleman is a project manager with Hewlett-Packard Corporate Engineering. He works in the area of software maintainability assessment and defect analysis.

Dan Ash is a firmware engineer for Hewlett-Packard Boise Printer Division. He specializes in font technology and embedded systems.

Bruce Lowther is a software engineer at Micron Semiconductor. He works in object-oriented development, focusing on software reusability and software quality. He is a member of the IEEE Computer Society.

Paul Oman is an associate professor of computer science at the University of Idaho where he directs the Software Engineering Test Lab. He is a member of the IEEE Computer Society.

Readers can contact the authors through Paul Oman, Software Engineering Test Lab, University of Idaho, Moscow, Idaho 83843, e-mail oman@cs.uidaho.edu.