SOFTWARE MAINTENANCE INTEGRATED WITH RELIAIBILITY

INTRODUCTION

For example, one study reports the following: About half of applications staff time was spent on maintenance, over 40% of the effort in supporting an operational application system was spent on user enhancements and extensions, and about half a man-year of effort was allocated annually to maintain the average system (1). In another report, the same authors list the factors that cause the significant maintenance effort: system age, system size, relative amount of routine debugging, and the relative development experience of the maintainers (2). System age drives the other factors: With increased system age, system size increases, leading to greater effort allocated to routine debugging; with increased system age, the relative development experience of the maintainers declines because of organizational turnover and change. All of these factors tend to increase the time and cost of performing maintenance. Thus maintenance, integrated with reliability, is an area that deserves a lot of attention. Improvements in maintenance practices should result in reduced costs and increased effectiveness of performing maintenance.

However, there is a limit to reducing cost and increasing effectiveness through improved practices, because the developer has largely determined the maintainability of the software before it ever reaches the maintainer. That is, its reliability has been determined. The maintainer can only influence reliability during the maintenance phase of the software life cycle. The reliability of the software as designed is determined, in part, by whether the software development methodology assists the developer in producing maintainable software. Consequently, maintenance practices, which maintainers control, and development methodology, which developers control, need to be standardized (3). The objective of standardization is to improve the maintainability of both existing and new software. One example of standardization is the IEEE Standard for Software Maintenance, IEEE 1219 (4). IEEE 1219 provides a process for managing and executing maintenance activities. Another example is ISO/IEC 12207, International Standard for Information Technology Software—Life Cycle Processes. The objectives of 12207 are to provide 1) a stable architecture for the software life cycle and 2) a common framework for world trade in software (5). However, the limitations of using standardization to solve the maintenance problem should be recognized.

In addressing the issue of integrating maintenance and reliability, it is useful to state what we expect of software. Four questions and their answers address this topic, using a hypothetical website example:

1. What *must* the software do (i.e., basic software reliability and maintainability requirements)?
   Consistently provide access to the user-designated websites.
2. What must the software *not* do (i.e., advanced software reliability and maintainability requirements)?
   Be impervious to change as the need develops to modify the initial design to incorporate features like security.
3. What *could* the software do (i.e., user expectations)?
   Consistently provide access to the user-designated websites and display *relevant information*.
4. What *does* the software do (i.e., operational experience)?
   If the user is lucky, it provides access to websites.

All questions are critical to meeting user needs, but questions 1 and 2 are particularly relevant from a reliability and maintainability perspective. Question 1 is related to, for example, providing high reliability under average load conditions and the capability to make minor software changes without introducing faults. Question 2, on the other hand, is related to, for example, providing high reliability under extreme load conditions and the capability to make major changes without causing catastrophic faults. Interestingly, if questions 1 and 2 are not satisfied, rather than achieving user goals, as in the answer to question 3, the user would be relegated to the unsatisfying answer to question 4!

APPROACHES FOR IDENTIFYING KNOWLEDGE REQUIREMENTS IN SOFTWARE MAINTENANCE AND RELIABILITY MEASUREMENT (6)

Two approaches exist to identifying the knowledge that is required to plan and implement a software reliability and maintenance measurement program. One approach is *issue-oriented*, as shown in Table 1. The other is *life cycle phase-oriented*, as shown in Fig. 1. The two approaches are compatible but are different views of achieving the same objective and have been provided to show the reader *why* (issue-oriented) and *when* (phase-oriented) the need for measurement occurs. A case study that addresses many of the issues and life cycle factors that we describe here can be found in a report on the NASA Space Shuttle software development and maintenance process (7).

Figure 1shows four phases of the software development cycle related to measurement, along with the documentation used in each phase and the metrics applicable to reliability and maintenance. *Static metrics* are those that are collected before the code is executed; *dynamic metrics* are collected when the code executes. In addition, as we move from left to right in the diagram, the metrics become progressively less qualitative and more quantitative because requirements documents are typically fuzzy whereas code listings, for example, are more definitive and can be subjected to quantitative analysis (e.g., complexity metrics can be computed).

Looking at Fig. 1, in the *analyze phase*, if reliability requirements are specified without considering the fact that all software is subject to change, the maintainability of the software will be at risk. For example, if the software is specified to have a predicted *time to next failure* far exceeding the mission duration, but says nothing about *time to next*

J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*. Copyright © 2007 John Wiley & Sons, Inc.

Table 1. Knowledge Requirements in Software Maintenance Measurement

Issue	Function	Knowledge
1. **Goals:** What maintenance goals are specified for the system?	**Analyze** maintenance goals and **specify** reliability	Reliability Engineering Requirements Engineering
2. **Cost and risk:** What is the cost of achieving maintenance goals and the risk of not doing so?	**Evaluate** economics and risk of maintenance	Economic Analysis Risk Analysis
3. **Context:** What application and organizational structure is the system and software to support?	**Analyze** the application environment	Systems Analysis Software Design
4. **Operational profile**: What are the criticality and frequency of use of the software components?	**Analyze** the software environment	Probability and Statistical Analysis
5. **Models:** What is the feasibility of creating or using an existing reliability model for assessment and prediction of maintenance, and how can the model be validated?	**Model** reliability and **validate** the model	Probability and Statistical Models
6. **Data requirements:** What data are needed to support maintenance and reliability goals?	**Define** data type, phase, time, and frequency of collection	Data Analysis
7. **Types and granularity of measurements:** What measurement scales should be used, what level of detail is appropriate to meet a given goal, and what can be measured quantitatively, qualitatively, or judgmentally?	**Define** the statistical properties of the data	Measurement Theory
8. **Product and process test and evaluation**: How can product maintenance and reliability measurements be fed back to improve process quality?	**Analyze** the relationship between product maintainability and reliability and process stability	Inspection and Test Methods
9. **Product Maintainability, Reliability and Process Quality Prediction**: What types of predictions should be made?	**Assess** and **predict** product maintainability, reliability, and process quality	Measurement Tools

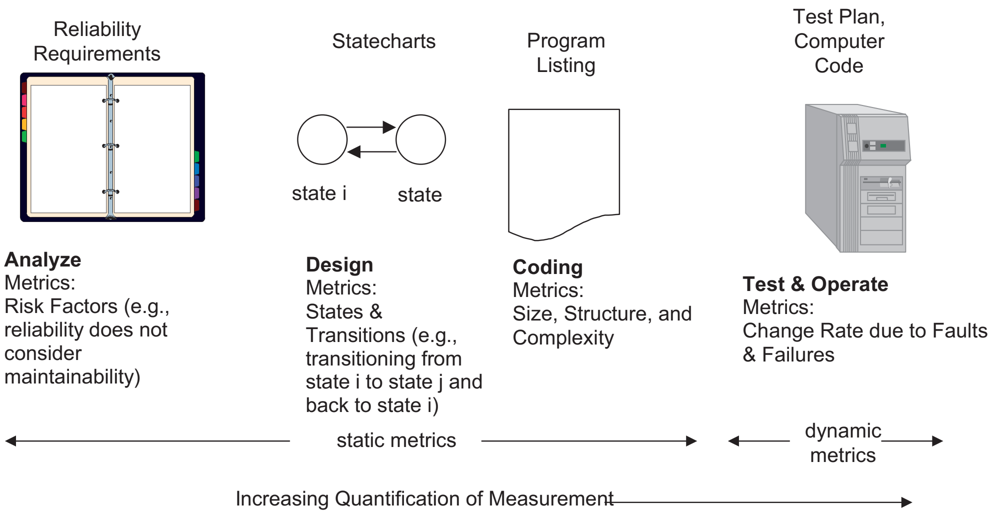

Figure 1. Life cycle measurement attributes.

failure requirement after the software has been changed, it is likely the mission would be jeopardized.

The second facet of Fig. 1that we need to consider is state transitions that should be identified during the *design phase*. An example is the state of the software when a fault is found (state *i* in Fig. 1), correcting the fault (state *j*), and returning to state *i* to find another fault. We would want the software designed so that rather than returning to state *j* from *i*, the software would transition to the *realistic* state *k* as the result of introducing a fault in the process of correcting one.

Third, in the *coding phase* of Fig. 1, our interest is in measuring the software with respect to *size* (e.g., source lines of code), *structure* (e.g., path count), and *complexity* [e.g., cyclomatic complexity (CC)]. Using the example of CC, we can say that it is a metric function $M = e - n + 2\,p$ whose inputs are number of edges e, number of nodes n, and number of connected components p in a directed graph representation of a program. The output of the function is a single numerical value M that is interpreted as the degree to which software possesses a given attribute (e.g., CC) that may affect its reliability (8). If reliability is adversely affected by excessive complexity, the implication is that the software would be difficult to maintain.

Finally, in the *test and operate phase* of Fig. 1, a primary concern is how *maintainable* the software will be when subjected to changes as a result of correcting for faults and failures. To do this task, one could examine the test plan to see whether it provides for regression testing (i.e., retesting everything in the code that could have been affected by faults and failures). In addition, the code would be scrutinized for fault proneness (i.e., the tendency for complex code to result in faults).

AN INTEGRATED APPROACH TO ANALYZING MAINTENANCE PROCESSES AND PRODUCT RELIABILITY

The relationship between product quality and process capability and maturity has been recognized as a major issue in software engineering based on the premise that improvements in processes will lead to higher quality products. An important facet of process capability is stability. Trend and change metrics across modules and within a module define and evaluate process stability. Our integration of product and process measurement serves the dual purpose of using metrics to assess reliability and risk *and* to evaluate process stability. We use the NASA flight software to illustrate our approach.

CONCEPT OF STABILITY

Trend Metrics

To gain insight about the interaction of the maintenance process with product metrics like reliability, two types of metrics are analyzed: *trend* and *change.* Both types are used to assess maintenance process stability *within* and *across* modules. By chronologically ordering metric values by module, defect, or change date, we obtain discrete functions in time that can be analyzed for trends. When analyzing trends, we note whether a trend is favorable (8). For example, a decreasing trend in defect count *D*, as a function *sloc* or CC would be favorable (i.e., D *decreases* as sloc and CC *decrease*).

Examples of Favorable Trends. Figures 2and 3show a favorable trend for *D versus sloc* and *D versus CC*, respectively. A favorable trend is indicative of maintenance stability because, in these cases, the beginning of the trend corresponds to the first module that is maintained and the end of the trend corresponds to the last module. Thus, as maintenance proceeds chronologically, with lower values of sloc and CC, the reliability of the maintained software increases.

In addition to using a plot to judge trend, we can use the correlation coefficient as a point estimate of the trend. These coefficients are shown below.

- 0.7418 D versus sloc
- 0.7342 D versus CC
- 0.8449 sloc versus CC

As sloc is highly correlated with CC (i.e., large software is complex and small software is less complex), it would be possible to use either sloc or CC, alone, as the trend indicator for defect count.

Examples of Unfavorable Trends. We collected and analyzed historical reliability data for *unfavorable* trends because we want to identify problems in development and maintenance that should be addressed. These data show in retrospect whether maintenance actions were successful in increasing (or decreasing) reliability based on a favorable (or unfavorable) trend. To this end, we determined whether our maintenance effort results in *decreasing* reliability *within a module* or over a *sequence of modules*. To do this task, we plot graphs of reliability metrics, such as *time to next defect within a module* and *defect density* (defect count / sloc) *across modules* to indicate whether the maintenance effort has been *unsuccessful* as it relates to reliability.

Figure 4shows the former case for a given Module 11181, where time to next defect is computed as $\Delta T_{i,\,I+1}$. Unfortunately, the trend does not support maintainability and stability because the trend is increasing $\Delta T_{i,\,I+1}$ as more defects are discovered for *this* module. There could be problems in development or maintenance, or both, with this module that should be investigated. The other modules should be subjected to the same analysis. Figure 5shows the latter case—trend across modules that is *unfavorable*. Recall that in Figs. 2and 3the tend was favorable across modules. The apparent contradiction with Fig. 5is explained by the fact that Fig. 5is based on *normalization* of defect count by sloc (i.e., defect density). The lesson learned from this exercise is that we should evaluate multiple metrics when assessing maintainability and reliability. In this example, we would recognize that different results could be obtained when module size is taken into account. Even one unfavorable trend, such as the one in Fig. 5, should lead us to question the effectiveness of our maintainability and reliability processes.

Change Metric

Although looking for a trend on a graph is useful, it is not a precise way of measuring stability, particularly if the graph has peaks and valleys and the measurements are made at discrete points in time. Therefore, we developed a Change

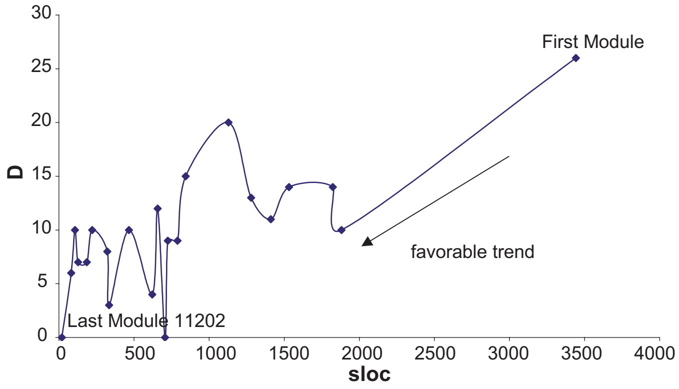

Figure 2. NASA flight software defect count D vs. source lines of code (sloc) by module.

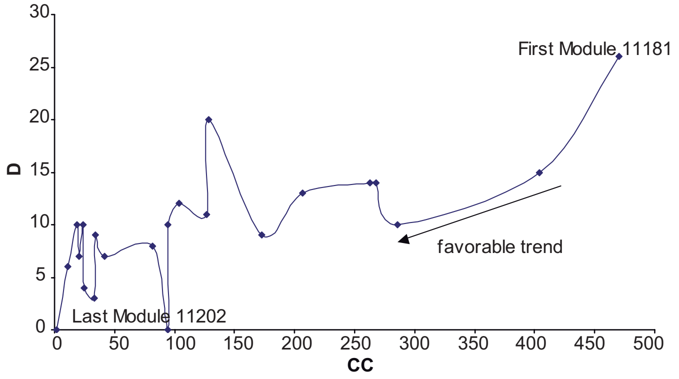

Figure 3. NASA flight software defect count D vs. cyclomatic complexity (CC) by module.

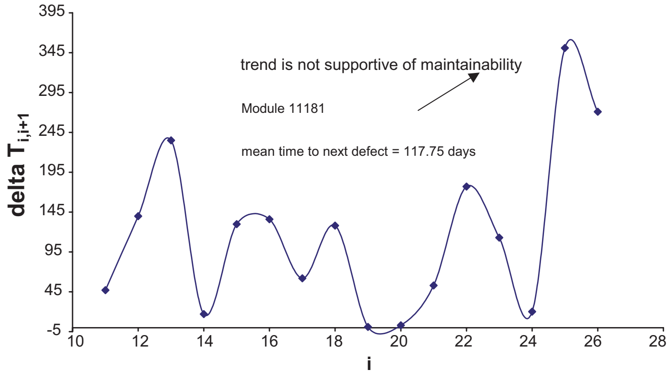

Figure 4. NASA flight software time to next defect (delta $T_{i,i+1}$) vs. defect i

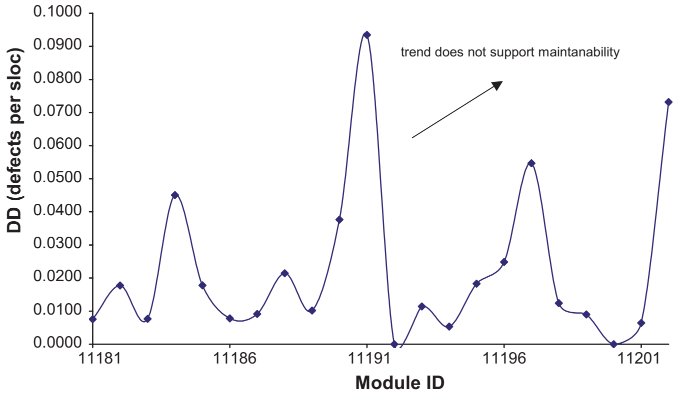

Figure 5. NASA flight software module defect density DD vs. module ID

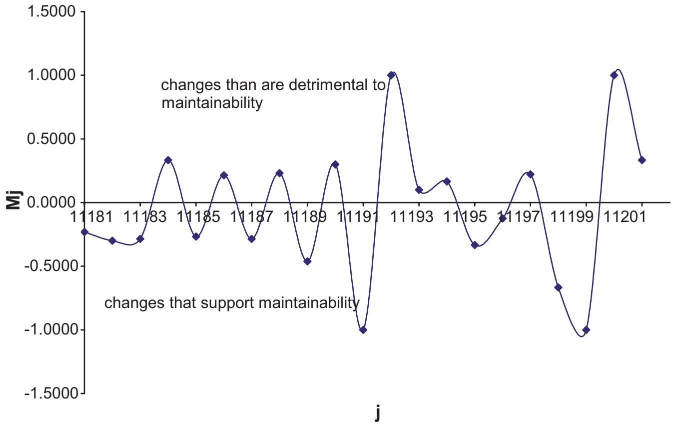

Figure 6. NASA flight software module change metric M_j (defect count) vs. module j

Metric (CM), which is computed as follows (8):

1 Compute the relative change in the metric from j to j + 1 (e.g., module j to module j + 1):

$$\begin{aligned}&(M_{j+1} - M_j)/M_j) \text{ if } M_j \geq M_{j+1}\\&(M_{j+1} - M_j)/M_{j+1}) \text{ if } M_j < M_{j+1}\end{aligned} \qquad (1)$$

2 Compute the mean of equation (1):

$$\overline{M} = \sum j = 1n((\mathrm{Mj} + 1 - \mathrm{Mj})/\mathrm{Mj}) \qquad (2)$$

3.a If $\overline{M}$ in equation 2 is *negative and $\overline{M}$ should be negative* (e.g., the mean of *defect count* changes is negative), then $\overline{M}$ implies that maintainability is getting better from j = 1 to j = n. Maintenance *stability* is indicated.

3.b If $\overline{M}$ in equation 2 is *negative and $\overline{M}$ should be positive* (e.g., the mean of *time to defect occurrence* changes is positive), then $\overline{M}$ implies that maintainability is getting worse from j = 1 to j = n. Maintenance *instability* is indicated.

4.a If $\overline{M}$ in equation 2 is *positive and $\overline{M}$ should be positive* (e.g., the mean of *time to defect occurrence* changes is positive), then $\overline{M}$ implies that maintainability is getting better from j = 1 to j = n. Maintenance *stability* is indicated.

4.b If $\overline{M}$ in equation 2 is *positive and $\overline{M}$ should be negative* (e.g., the mean of *defect count* changes is negative), then $\overline{M}$ implies that maintainability is getting worse from j = 1 to j = n. Maintenance *instability* is indicated.

5 $\overline{M}$ is the CM in the range $-1, 1$. The numeric value of CM indicates the degree of maintenance stability or instability.

Now we compute various change metrics using the NASA flight software—defect count and size and complexity metrics—computed from the NASA maintenance activity. An example is shown in Fig. 6where the defect count CM is plotted against module *j*. Increases in *j* represent chronologically increasing module maintenance activity. Values of M_j below the horizontal axis represent changes that support maintainability; those above are detrimental to maintainability. With this type of plot, software engineers can see how maintainability changes in going from module *j* to *j + 1*. For example, in transitioning from module 11183 to 11184, a detrimental change is induced (i.e., positive), whereas transitioning from module 11184 to 11185 induces a supportive change (i.e., negative). The values of $\overline{M}$—the CM—is equal to -0.0502, -0.0988, and -0.1330 for *defect count*, *sloc*, and CC, respectively. As all metrics are negative, the implication is that maintenance activity is stable. Furthermore, the fact that the CC CM is the most negative suggests that complexity is a key factor in achieving maintainable software.

CONCLUSIONS

Our emphasis in this article was to propose a unified product and process measurement model for both product evaluation and process stability analysis. We were less interested in the results of the NASA flight software stability analysis, which were used to illustrate the model concepts. We conclude, based on retrospective use of reliability, risk, and maintenance metrics, embodied in trend and change metrics, that it is feasible to measure and assess both product quality and the stability of a maintenance process. The model is not domain-specific. Different organizations may obtain different numerical results and trends than the ones we obtained for the NASA data.

BIBLIOGRAPHY

1. Lientz, B. P.; Swanson, E. B., Problems in Application Software Maintenance. *Commun. ACM* 1981, **24**(11),pp 763–769.
2. Lientz, B. P.; Swanson, E. B., *Software Maintenance Management*. Addison-Wesley: Reading, MA, 1980.
3. Schneidewind, N. F., Software Maintenance: The Need for Standardization. *(this paper has been translated into Russian and published in the Soviet edition of this journal). Proc. IEEE* 1989, **77**,pp 618–624.
4. IEEE Standards. IEEE Standard for Software Maintenance, IEEE Std 1219-1993. Institute of Electrical and Electronics Engineers: New York, 1993.
5. Pigoski, T. M., *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. Wiley: New York, 1997.

6. Schneidewind, N. F., *Body of Knowledge for Software Quality Measurement*. IEEE Computer, Computer Society Press, Los Alamitos, CA, February 2002, pp 77–83.
7. Billings, C., *et al.* Journey to a Mature Software Process. *IBM Syst. J.* 1994, **33**(1),pp 46–61.
8. Schneidewind, N. F., Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics. *IEEE Trans. Software Eng.* 1999, **25**(6),pp 768–781.

NORMAN F. SCHNEIDEWIND
Fellow IEEE