Measurements to Manage Software Maintenance

George E. Stark
The MITRE Corporation

Software maintenance is central to the mission of many organizations. Thus, it is natural for managers to characterize and measure those aspects of products and processes that seem to affect cost, schedule, quality, and functionality of a software maintenance delivery. This article answers basic questions about software maintenance for a single organization and discusses some of the decisions made based on the answers. These examples form a useful basis for other software maintenance managers to make better decisions in the day-to-day interactions that occur on their programs.

Software maintenance, the process of modifying existing operational software, is a critical topic in today's industry [1]. Maintenance consumes between 60 percent and 80 percent of a typical product's total software lifecycle expenditures [2-4], and over 50 percent of programmer effort is dedicated to it [5]. Given this high cost, some organizations view their maintenance processes as an area in which to gain a competitive advantage [6].

Several authors note that maintenance of software systems intended for a long operational life pose special management problems [7-9]. The Software Engineering Institute (SEI) believes that organizational processes greatly affect the predictability and quality of software [10], while J. Arthur and R. Stevens explain the importance of documentation to software maintenance [11]. Additionally, M. Hariza et al., B. Curtis, and C. Yuen conclude that programmer experience is at least as important as code attributes in determining software maintenance complexity [9,12,13]. Thus, software maintenance planning and management should be formalized and quantified.

In 1994, the Missile Warning and Space Surveillance Sensors (MWSSS) program management office took over the maintenance effort on the seven products upon which this article is based. Executing in 10 locations worldwide, these products total 8.5 million source lines of code written in 22 different languages. Some systems are more than 30 years old; the newest system became operational in 1992. To understand and manage the software maintenance effort, we instituted a measurement program based on V. Basili's Goal-Question-Metric paradigm [14] and other reported measurement experiences [15-19, 23]. Our goals are to improve customer satisfaction and meet our commitments to cost and schedule. The program questions relate to the size and type of workload we handle, the amount of wasted effort, the cost and schedule of a release, and the quality of the release. Table 1 contains the results.

Goal	Question	Metric(s)
Maximize Customer Satisfaction	How many problems affect the customer	Current Change Backlog
		Software Reliability
	How long does it take to fix an Emergency or Urgent problem?	Change Cycle Time from Date Approved and from Date Written
Minimize Cost	How much does a software maintenance delivery cost?	Cost per Delivery
	How are the costs allocated?	Cost per Activity
	What kinds of changes are being made?	Number of Changes by Type
	How much effort is expended per change type?	Staff Days Expended/Change by Type
	How many invalid change requests are evaluated?	Percentage of Invalid Change Requests Closed Each Quarter
Minimize Schedule	How difficult is the delivery?	Complexity Assessment
		Software Maintainability

		Computer Resource Utilization
	How many changes are made to the planned delivery content?	Percentage Content Changes by Delivery
	Are we meeting our delivery schedules?	Percentage of On-time Deliveries

Table 1. ***Software maintenance goals, questions, and metrics.***

Notice that the questions can support more than one goal. For example, the question "Are we meeting our delivery schedules?" supports the goal of customer satisfaction as much as the goal to minimize schedules. For brevity, Table 1 shows only one occurrence.

Engineers and managers in our organization use the metrics information three ways:

- Direct attention - what problems do they need to address?
- Solve problems - which choice should they make?
- Keep score - how are they doing?

This article examines software maintenance in this context, summarizing the MWSSS software maintenance process, showing which process and product changes are under consideration, and describing some limitations to our measurement program.

The Software Maintenance Process

Figure 1 depicts the MWSSS software maintenance process. Using a problem report, a user identifies a desired change, e.g., a change in requirements or a need to fix an improper function. An analyst at the user's location reviews the problem report for completeness, checking it against known problem reports to eliminate duplication and to determine if the user correctly understood the system. If the problem was previously unknown but correctly identified, the analyst categorizes the problem report as either a software, hardware, or communications equipment problem. For software problems, a software change form (SCF) is generated. The analyst completes 18 SCF data items, including dates for submission and approval, current method, proposed change, justification, and resource estimates.

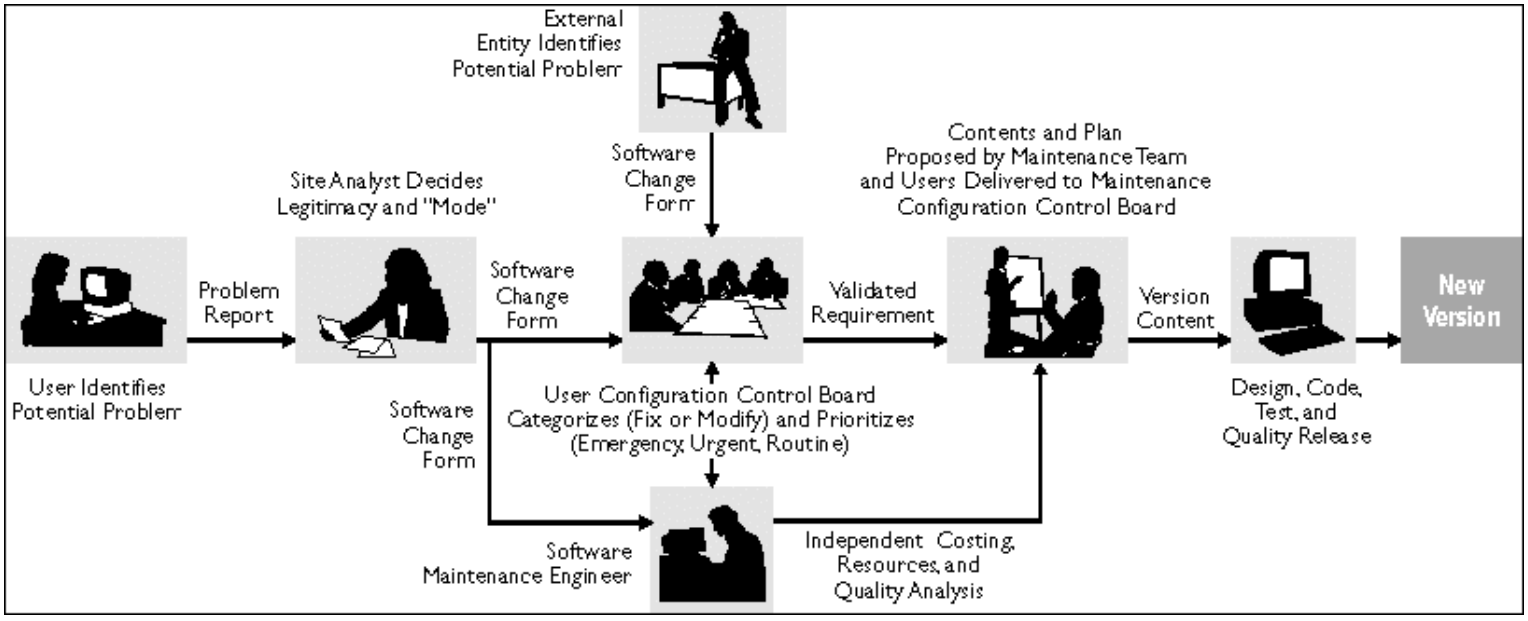

Figure 1. *Software maintenance process.*

The analyst then sends the completed SCF to a user board for validation and to a software maintenance engineer for independent evaluation; this maintenance engineer then evaluates the SCF based on three criteria: (1) effort required to complete the change, (2) computer resources required, and (3) impact on the quality of service. The engineer makes the effort estimate based on the taxonomy of change types described later. If this independent analysis is 20 percent greater or less than original user's estimates, the user and engineer meet to resolve the difference. This often helps clarify the requirement.

Another interfacing system or the maintenance team also can generate SCFs that affect systems for which they are not directly responsible but with which they must interface. These may result from technical initiatives. For example, if system A is upgrading its communication protocol and it exchanges information with system B, people from system A can generate an SCF for system B to upgrade its communications protocol.

Next, the user board categorizes the SCF as either a modification or a fix. A modification is a change that is not a part of the current system requirements. A fix is a fault correction.

The board also recommends a priority to the SCF: Emergency if the SCF is necessary to avoid system downtime or if high-priority mission requirements dictate; Urgent if the SCF is needed in the next software delivery for an upcoming mission or to fix a problem that arose as a result of a change in operations; Routine for all other SCFs, e.g., unit conversions, default value changes, and to fix printout.

Next, the SCF is queued for incorporation into a version release. The users, maintenance team, and system engineering organization then negotiate the version content. This planning and approval process includes a review of several metrics: release complexity, software reliability, software maintainability, and computer resource utilization. The maintenance configuration control board then reviews the release plan and schedules the release. The software engineering team then completes the design, code, test, installation, and quality assurance of the release, with user and system engineering reviews at milestone dates throughout the release.

The metrics in Table 1 reflect the four information products in Figure 1 over which our management team has control: requirements validation, independent costing, delivery content definition, and release implementation. The earlier phases in the process, although important, have little effect on our organization; thus, we do not measure them.

Maximize Customer Satisfaction

Customer surveys and interviews revealed that our customer is satisfied when few system problems affect their ability to do their job, complaints are resolved quickly, and the supplier meets its commitments. The metrics that follow address these issues.

How Many Problems Affect the Customer?

We track the number of unresolved customer complaints over time. Figure 2 shows the backlog of change requests over the past two years for one project, plus the planned and actual release content. The triangles represent the planned number of changes for a release, the bars represent the actual number of changes closed, and the squares represent the total backlog of change requests at the end of the month. Managers use this chart to allocate computer and staff resources, plan release content, and track the effect of new tools or other process improvement programs over time.

If the backlog remained at zero, a manager would not conclude that no problems are affecting the user; it instead may indicate that their office is overstaffed or in equilibrium with the incoming change requests. G.E. Stark, L.C. Kern, and C.W. Vowell describe an alternative to this metric, which looks at the change request arrival and closure rates [17].

Figure 2 also demonstrates how well the organization is meeting schedule commitments. It shows that we made eight of the last 10 deliveries on schedule.

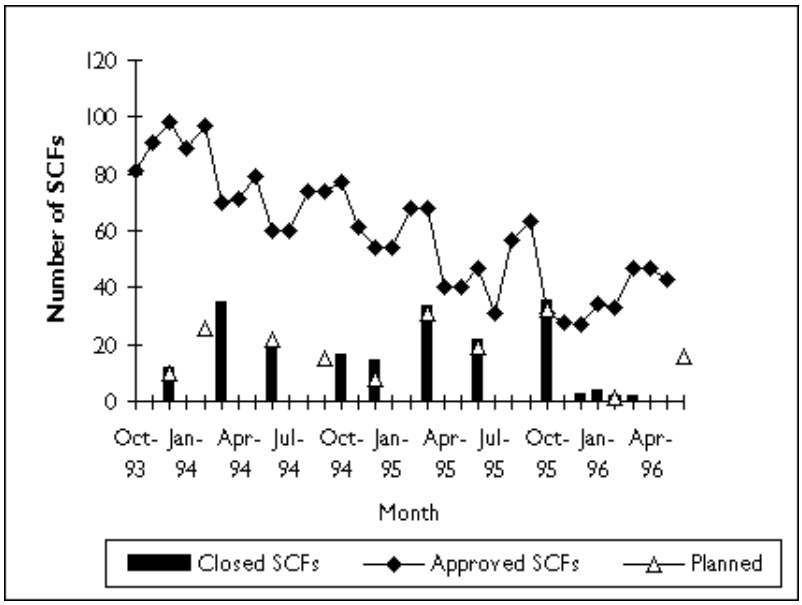

Figure 2. *Backlog of change requests, release plan, and actuals by month.*

Software Failure Rate

We also track the software failure rate in the field to understand how many problems are affecting the customer. A failure is charged whenever the system performs below user requirements. The site analyst and engineer inspect each problem report to determine if a software failure occurred. All downtime incidents (reported in monthly maintenance logs) caused by a software failure are counted. Figure 3 shows the failure rate for the past nine deliveries of one product. Over the nine releases, the maintenance team has reduced the number of operational failures from 6.8 per 1,000 operational hours to fewer than 2.4 per 1,000 hours. This improvement in product quality shows up in the backlog chart of Figure 2. Users generate fewer change requests when the system operates without failure for longer periods.

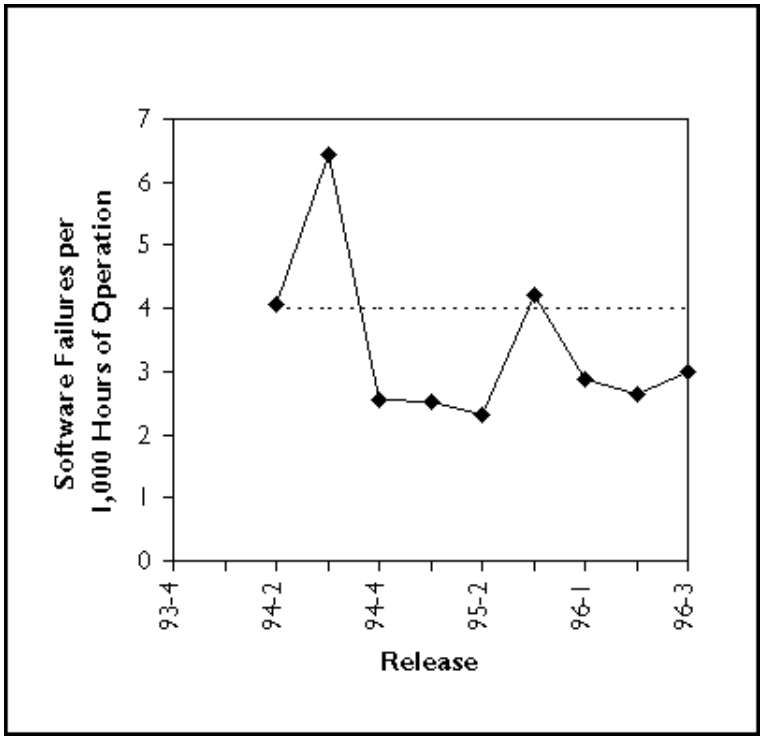

Figure 3. *Software failure rate by delivery.*

The system manager uses the failure rate information in several ways. First, system managers set a threshold at four failures per 1,000 hours of operation. If the system is operating far below this threshold, the manager may decide to incorporate more difficult changes into the next release. If the system is operating above the threshold, the decision may be to switch to a previous version of the system or to only allow fault correction and no modifications until the system is again below the threshold.

Second, we use the graph to estimate the expected number of events during a mission and the probability of completing a mission of certain duration without an event. For example, if the customer is planning a one-week mission and needs to know the probability of the software delivery supporting the entire mission, the probability is given by

Pr[no fail in wk] = exp(-168 hrs/wk*.002 fails/hr) = 0.71

or a 71 percent chance that the system will not fail in one week of operation because of a software problem.

Finally, we use the graph in Figure 3 to establish future system requirements with the customer. We have used the historical failure rate as a quality requirement to trade off the cost and schedule against when the customer requests a major system upgrade. J.D. Musa, et al., describe the approach well [20].

How Long Does It Take to Fix a Priority Problem?

This is another question related to customer satisfaction. Figure 4 helps gauge satisfaction with a cumulative distribution graph of two important process durations. The line with square markers represents the percentage of priority changes delivered within a given number of days from the time the user writes the change request. The line using diamond markers represents the time to deliver a priority change after the requirement has been approved by the using community.

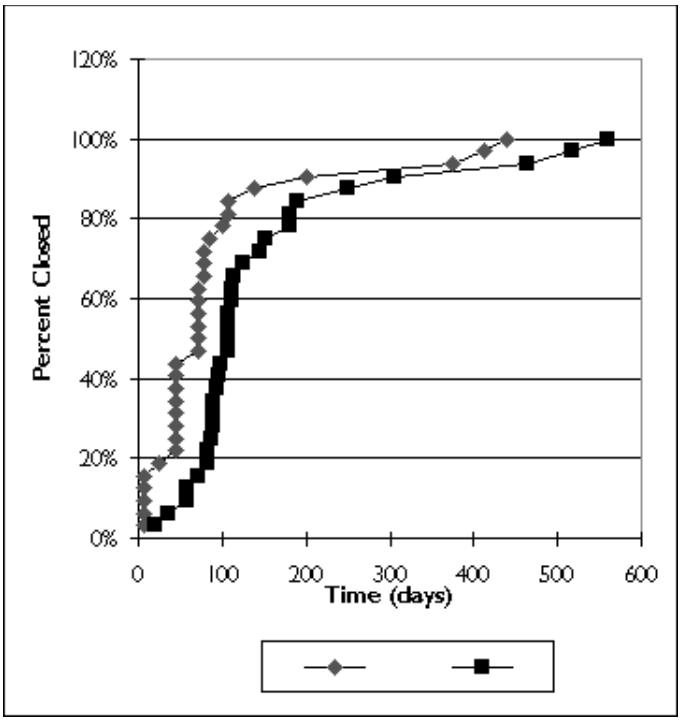

Figure 4. *Software change form cycle times.*

The horizontal distance between the two lines on the chart is the "in-process" time, i.e., the time the change spends in board review. For this system, the in-process time averaged 54 days. The chart also shows that approximately 80 percent of our urgent change requests are installed and ready for use within 90 days of user board approval or within 170 days from when they are written by the customer.

Thus, these three simple graphs tell us how many problems are affecting the customer, whether we are meeting our schedule commitments, and how long it takes to fix a priority problem.

Minimize Cost

Many people participate in the software maintenance process. The 11 cost categories of software maintenance are software development activities, configuration management, quality assurance, security, administrative support, travel, project management, system engineering, hardware system maintenance, system management, and finance. These costs are further broken into categories of common and system-specific costs. Hardware maintenance is a system-specific cost since each of our systems uses different hardware platforms requiring different levels of hardware maintenance. To ease accounting, we apportion travel to a release rather than an activity.

The remaining categories are common costs across our organization. Some people participate in more than one activity, and many activities do not require full-time staff attention. We allocate the common costs to each release as a cost per day.

How Much Does a Delivery Cost?

By measuring the cost, each activity contributes to the total release cost, managers can direct their cost reduction efforts in certain areas. For example, Figure 5 shows the per release cost for 17 deliveries. The average cost for this set is $373,500 with a standard deviation of $231,700. We use these high-level statistics for long-range (two to five years) budget planning.

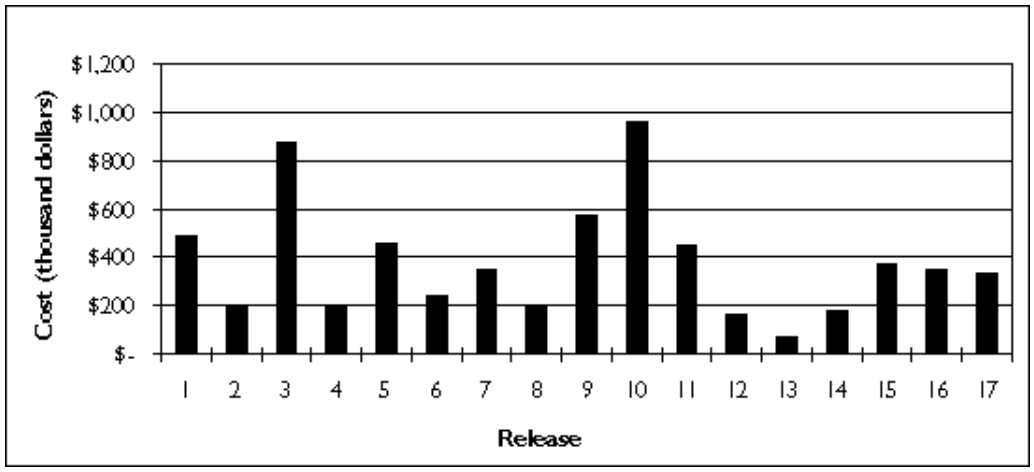

Figure 5. *Cost per delivery for one year.*

How Are the Costs Allocated?

For a typical release, the cost associated with the software development activities (design, code, unit test, and integration test) and hardware system maintenance make up 88 percent of the total release cost. The other nine cost categories account for the remaining 12 percent. This is encouraging because it shows that most of our money is going to the productive work of producing a release. We would, however, like to reduce the total cost of a release. Thus, the hardware system maintenance cost is a target of opportunity. We are examining the root causes of the hardware maintenance cost to find areas for improvement.

What Kinds of Changes Are Being Made?

To answer this question, we developed the software change taxonomy shown in Table 2. It includes 10 types of changes and root causes for each change type.

Computational
Incorrect operand in equation Incorrect use of parentheses Incorrect/inaccurate equation Rounding or truncation error
Logic
Incorrect operand in logical expression Logic out of sequence Wrong variable being checked Missing logic or condition test Loop iterated incorrect number of times
Input
Incorrect format Input read from incorrect location End-of-file missing or encountered prematurely
Data Handling
Data file not available Data referenced out-of-bounds Data initialization Variable used as flag or index not set properly

Data not properly defined/dimensioned Subscripting error
Output
Data written to different location Incorrect format Incomplete or missing output Output garbled or misleading
Interface
Software/hardware interface Software/user interface Software/database interface Software/software interface
Operations
COTS/GOTS software change Configuration control
Performance
Time limit exceeded Storage limit exceeded Code or design inefficient Network efficiency
Specification
System/system interface Specification incorrect/inadequate Requirements specification incorrect/inadequate User manual/training inadequate
Improvement
Improve existing function Improve interface

Table 2. ***Software change taxonomy.***

We categorized the changes delivered in the last eight releases using this taxonomy. This consisted of 67 modification (38 percent) and 110 fix (62 percent) changes. Figure 6 is a Pareto diagram of this change data. The left vertical axis shows the actual number of changes attributed to each class, the right vertical axis represents the cumulative percentage of defects, making it a convenient scale from which to read the line graph. The line graph connects the cumulative percents (and counts) at each category.

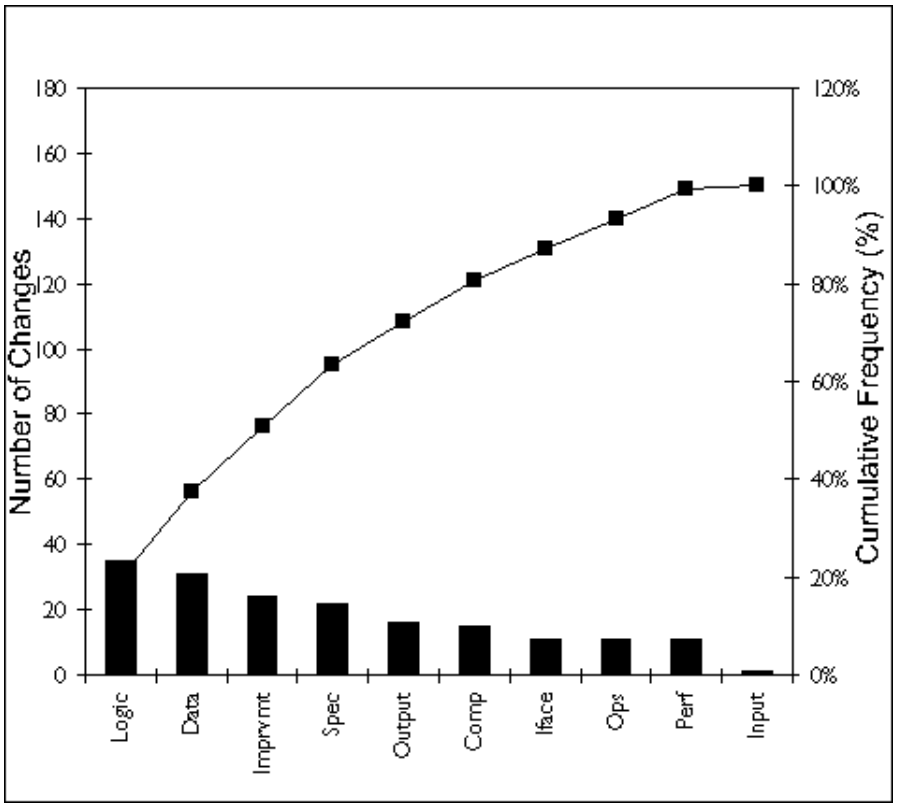

Figure 6. *Software maintenance changes by type.*

How Much Effort Is Expended per Change Type?

Figure 7 is a Pareto diagram of the effort required to make each change. This figure shows that although changes based on requirements or interface specification changes ranked fourth in number of changes with 22, they account for 42 percent of the total effort at 582 staff-days. Logic changes fall to third in effort when viewed in this manner.

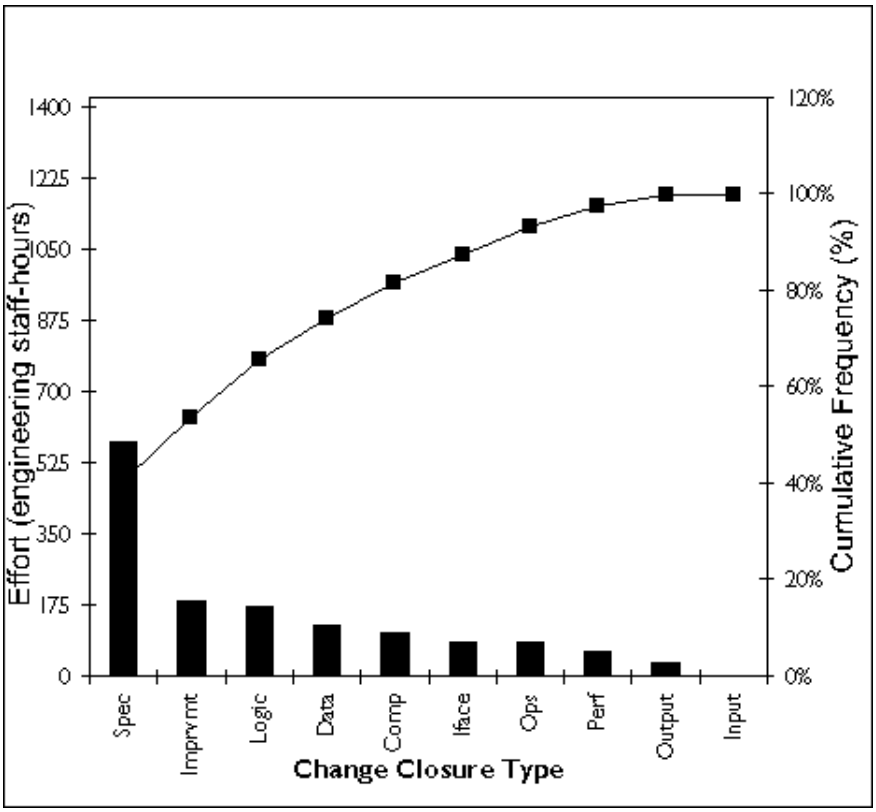

Figure 7. *Staff-days effort by category.*

By reviewing change requests and accurately assigning them to the change taxonomy, we can estimate the engineering staff-days required to design, code, and test individual changes. For example, the average staff-days of effort due to interface specification changes are 36 staff-days, with a standard deviation of 43 staff-days, whereas the average due to requirements specification changes is 22 days, with a standard deviation of 24 staff-days. We combine this information with the distribution of expenses to allocate activity costs. Next, we track the actuals against the estimate, updating the taxonomy and cost information as each release is completed.

How Complex Is the Delivery?

We agree with B. Curtis that complexity is "a loosely defined term" [21] that is usually concerned with understanding the structure of the code. Several measures with this goal in mind have been proposed [22-25]. We believe that the complexity of a maintenance release involves much more than just the code. Hence, we define complexity as *the degree to which characteristics that affect maintenance releases are present.*

"Characteristics that affect" include product characteristics (such as age, size, documentation, criticality, performance), management processes (such as abstraction techniques, verification and validation, maturity, tools), staff experience (such as system, scheduling, group dynamics), and the environment (such as office space, tools, target system compatibility).

We evaluate the complexity of a proposed set of changes using the spreadsheet-based tool described in [26]. The tool calculates a release complexity between 0 (lowest) and 1 (highest) based on a set of objective and subjective data supplied by the system engineer. After using the tool for one year, we found we are more likely to meet our cost, schedule, and quality targets if we can take action to keep the release complexity below a 0.5 value. Thus, for each release proposal (before proposing the delivery contents and plan to the Maintenance Configuration Control Board), software engineers and managers evaluate the results of the tool and, if necessary, implement complexity-reducing techniques, e.g., less content, more experienced staff, more rigorous process, and better tools.

For example, Figure 8 shows the results of a delivery complexity analysis. In this case, the first use of the complexity tool helped the manager recognize that the product was driving delivery complexity. The delivery had more changes than previous releases, affected many modules, and changed external interfaces. The changes involved many different languages and consisted of manipulating the real-time exception processing. The ease and completeness of test diagnostics were a concern as was the difficulty of installation for the release.

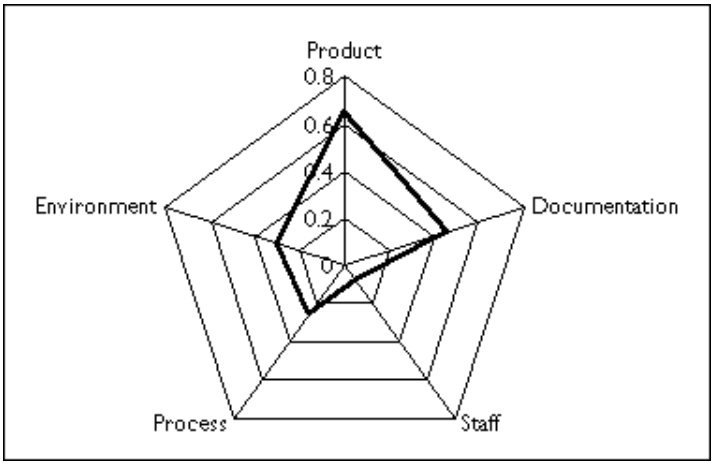

Figure 8. *Software delivery complexity drivers.*

To mitigate this product complexity, the manager assigned his best team to the release. (Although this approach would possibly be frowned on by the SEI because it is not a process focus, it is a typical decision in most software organizations.) During the second iteration, the manager found that the overall release complexity was still above the 0.5 rule of thumb and that process and environment were driving factors (the goal was to include all the product changes as a single release). Thus, the manager enforced a good process and made sure the team had access to a maintenance environment compatible with the target system that had tools to support the maintenance effort.

The combination of the above decisions shifted the complexity focus to the documentation. The decisions also reduced the overall complexity to a typical level before the delivery was implemented. The project was delivered on schedule, less than 5 percent over budget, and with no reported defects during a user trial period.

How Many Invalid Change Requests Are Evaluated?

At any time during the software change process, the user board can withdraw an SCF as invalid. An approved SCF may be withdrawn based on another change that corrects the problem or based on some external event, e.g., site closing or external interface change. The percentage of change requests that are withdrawn out of the total evaluated is a measure of rework in our process. Figure 9 shows the data for six quarters. On average, we evaluate 72 change requests each quarter. Of this, an average of 8 percent are withdrawn, which translates to a loss of only $7,500 per year (5.5 hours per evaluation * 5.75 rework evaluations per quarter * $60 per hour * four quarters per year).

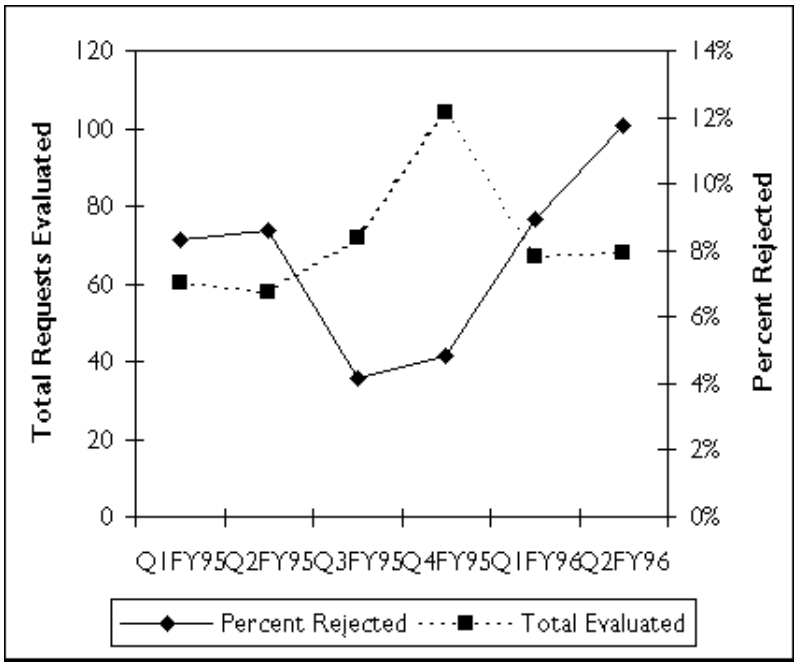

Figure 9. *Percent of rework by quarter.*

How Many Changes Are Made to the Planned Delivery Content?

Once we agree on a delivery plan with the customer, requirement volatility becomes a major factor. Requirement volatility comes in three types: additions to the delivery content, deletions from the delivery content, and changes in scope to a requirement. The requirement volatility for 19 deliveries is shown in Figure 10. The y-axis represents the percentage of planned requirements changed during the release. The graph follows a sequential order; thus, our more recent deliveries have had far less requirements volatility than the earlier releases, partially because of the focus on this measurement.

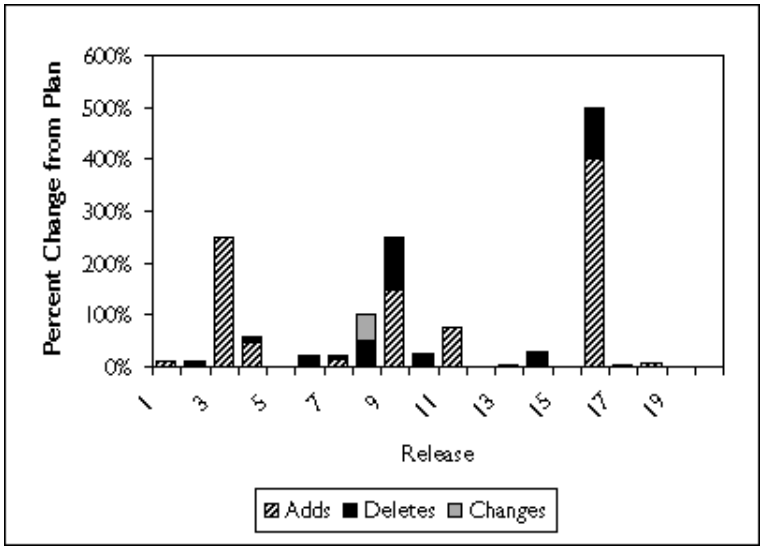

Figure 10. *Requirements volatility for 19 deliveries.*

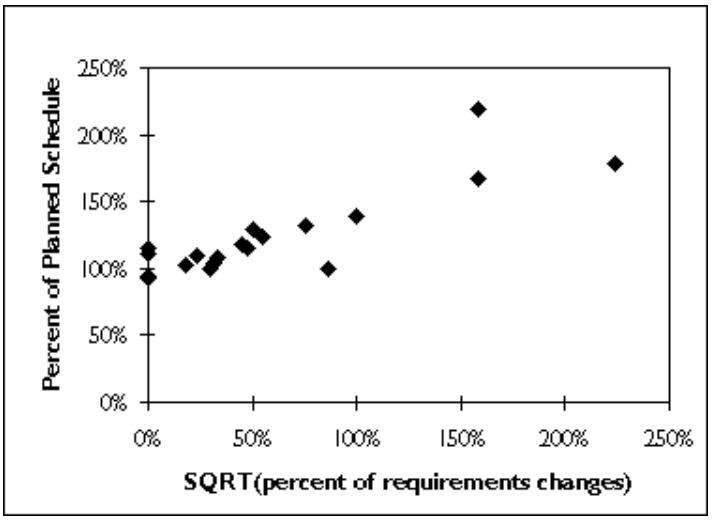

Figure 11. *Schedule volatility vs. requirements volatility.*

Figure 11 is a scatter plot of schedule performance vs. the square root of the percentage requirement volatility for these 19 deliveries. We chose the square root transformation to increase the contribution of values less than 100 percent and decrease the contribution of values much greater than 100 percent. A 100 percent value means the schedule was met, numbers less than 100 percent indicate early delivery, and numbers greater than 100 percent indicate late delivery. We developed a linear curve fit (least squares) to predict schedule volatility using this data combined with delivery effort data from Figure 7. The model is of the form

Percent Schedule = 0.97 + 0.41*(percent rqmt vol)$^{1/2}$ + 0.23*(changes/staff-day)

This equation yields a coefficient of correlation (R^2) of 0.72 with a standard error of 0.17. This implies that the model approximates the schedule slip fairly well. Notice that the schedule change goes up regardless of whether the requirement's change was an addition or deletion because the input to the model is a percentage of the requirement's change. This is a topic of debate in the organization. Some people argue that removing requirements involves effort to change the design and test procedures; others argue that a reduction in requirements means less work for the team and should result in completing the project in less time.

The equation explains the expected impact of changes to the delivery plan as they arise. For example, one version contained 15 planned requirements scheduled for delivery in 91 calendar days. However, the customer wanted to drop two of the requirements and change the scope of a third at preliminary design. Managers estimated a change in risk to version delivery of between 0.14 (15 changes in 108 staff-days) and 0.1 (13 changes in 130 staff-days). Note that the change in scope on one requirement is captured in the staff-days of the model rather than the number of requirements. Using the model, managers forecasted the impact to be $\mathbf{[0.97 + 0.41*(0.2)^{1/2} + 0.23*(0.1)] = 1.18}$ or an 18 percent schedule slip or 16 days added to the 91-day schedule.

Upon seeing the model and prediction, the customer decided that this slip was not acceptable and decided to incorporate only the scope change. The metrics-based model facilitated objective communication about version release plans and benefited customer relations.

Limitations

Although our measurement program provides benefits, it has three major limitations. First, we do not have an acceptable way to measure and manage software maintainability. This is a concern because cumulative software changes can make the code "brittle" (less structured). We have tried to quantify the maintainability of our software by evaluating the impact of releases with the survey-based approach proposed by the Air Force Operational Test and Evaluation Center (AFOTEC) [27] and the polynomial approach proposed by P.W. Oman, et al. [28]. Neither of these approaches have proven repeatable across our systems.

Second, we do not have a method to predict the computer resource impacts of individual change requests. Many of our systems are old and require constant capacity planning to keep the CPU below 100 percent busy, memory below 98 percent full, and some free disk space. Estimating the impact of individual modification requests on our computer resources is an important but currently impossible task.

Finally, our metrics program does not measure size in a way comparable to other programs. We measure size by the number of changes completed rather than by source lines of code (SLOC) changed or function points delivered, even though individual changes vary greatly in size and complexity (as evidenced by Figure 7). By comparing our productivity and process throughput with other organizations or "industry averages," which are normalized by SLOC or function points, we could look at alternative process improvements.

Summary

Although our measurement program is just over two years old, we have reaped many benefits and shown how measurements enhance software maintenance management. We began with clearly defined goals targeted toward project managers and engineers. We asked questions that assessed the current state of the product and process to predict future states.

Metrics analysis begins with insight into the workings of our software maintenance processes. It continues with calculations from conceptual models that reflect that insight and results in answers to the original questions. Managers direct their attention, solve problems, or keep score over time based on the answers. This article describes several decisions and examines the limitations of our program. I hope these examples form a useful basis for other software maintenance managers to make better decisions in the day-to-day interactions and long-term process improvement initiatives that occur on their programs.

About the Author

George Stark is a principal scientist with the MITRE Corporation in Colorado Springs, Colo., where he supports the software efforts of the Missile Warning and Space Surveillance Sensors Program Management Office. His technical interests include software metrics and reliability for management decision making. He has been involved in software reliability measurement for 13 years and was the vice chairman of the American Institute of Aeronautics and Astronautics blue-ribbon panel on software reliability. He has been the manager of software testing and reliability for a local loop fiber optic telephone system. He received the Johnson Space Center Quality Partnership Award and the MITRE General Manager's Award for contributions to software measurement. He received a bachelor's degree in statistics from Colorado State University and a master's degree in mathematics from the University of Houston.

The MITRE Corporation
1150 Academy Parl Loop, Suite 212
Colorado Springs, CO 80910
Voice: 719-572-8468
Fax: 719-572-8345
E-mail: gstark@mitre.org

References

1. Boehm, B., *Software Engineering Economics*, Prentice-Hall, New York, 1981.
2. Lientz, B. P. and E. B. Swanson, "Characteristics of Application Software Maintenance," *Communications of Association for Computing Machinery*, June 1978, pp. 466-471.
3. Parikh, G., *The Guide to Software Maintenance*, Winthrop Publishers, Cambridge, Mass., 1982.
4. Yau, S. S. and T. J. Tsai, "A Survey of Software Design Techniques," *IEEE Transactions on Software Engineering*, June 1986, pp. 713-721.
5. Gibson, V. and J. Senn, "System Structure and Software Maintenance Performance," *Communications of Association for Computing Machinery*, March 1989, pp. 347-358.
6. Moad, J., "Maintaining the Competitive Edge," *Datamation*, February 1990, pp. 61-66.
7. Card, D. N., D. V. Cotnoir, and C. E. Goorevich, "Managing Software Maintenance Cost and Quality," *Proceedings of International Conference on Software Maintenance*, 1987.
8. Chapin, N., "The Software Maintenance Life-Cycle," *Proceedings of International Conference on Software Maintenance,* 1988.
9. Hariza, M., J. F. Voidrot, E. Minor, L. Pofelski, and S. Blazy, "Software Maintenance: An Analysis of Industrial Needs and Constraints," *Proceedings of International Conference on Software Maintenance*, Orlando, Fla., 1992.
10. Software Engineering Institute, "Software Process Maturity Questionnaire Capability Maturity Model, Version 1.1," Carnegie Mellon University, Pittsburgh, Pa., 1994.
11. Arthur, J. and K. Stevens, "Assessing the Adequacy of Documentation Through Document Quality Indicators," *Proceedings of International Conference on Software Maintenance,* Miami, Fla., 1989.
12. Curtis, B., "Conceptual Issues in Software Metrics," *Proceedings of IEEE International Conference on System Sciences*, 1986.
13. Yuen, C., "An Empirical Approach to the Study of Errors in Large Software under Maintenance," *Proceedings of International Conference on Software Maintenance,* 1985, pp. 96-105.
14. Basili, V. and H. D. Rombach, "Tailoring the Software Process to Goals and Environments," *Proceedings of 9th International Conference on Software Engineering*, Monterey, Calif., 1987.
15. Grady, R. B., "Measuring and Managing Software Maintenance," *IEEE Software*, April 1987.
16. Stark, G. E., R. C. Durst, and C. W. Vowell, "Using Metrics for Management Decision-Making," *IEEE Computer*, September 1994.
17. Stark, G. E., L. C. Kern, and C. W. Vowell, "A Software Metric Set for Program Maintenance Management," *Journal of Systems and Software*, Vol. 24, 1994, pp. 239-249.
18. Grady, R. B., *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs, N.J., 1992.
19. Lydon, T., M. Wall, and L. Fischer, "Software Metrics at Raytheon," *Proceedings of 4th Annual Oregon Workshop on Software Metrics*, Silver Falls, Ore., March 1992.
20. Musa, J. D., A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, and Application*, McGraw-Hill, New York, 1987.
21. Curtis, B., "Measurement and Experimentation in Software Engineering," *Proceedings of IEEE*, Vol. 68, No. 9, pp. 1144-1157.
22. McCabe, T., "A Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2, 1976, pp. 308-320.
23. Fenton, N., *Software Metrics: A Rigorous Approach*, Chapman and Hall, London, England, 1991.
24. Kafura, D. and G. R. Reddy, "The Use of Software Complexity in Software Maintenance," *IEEE Transactions on Software Engineering,* Vol. SE-13, No. 3, 1987.
25. Card, D. N. and R. L. Glass, *Measuring Software Design Quality*, Prentice-Hall, Englewood Cliffs, N.J., 1990.
26. Stark, G. E. and P. W. Oman, "A Survey Instrument for Understanding the Complexity of Software Maintenance," *Software Maintenance: Research and Practice*, December 1995, pp. 421-441.
27. AFOTEC, *Software Maintainability Evaluation Guide*, AFOTEC Pamphlet 800-2, Vol. 3, HQ Air Force Operational Test and Evaluation Center, Kirtland Air Force Base, N.M. 87117-7001.
28. Oman, P. W., D. Ash, J. Alderete, and B. Lowther, "Using Software Maintainability Models to Track Code Health," *Proceedings of International Conference on Software Maintenance*, 1994, pp. 154-160.