Measurements for Managing Software Maintenance

George E. Stark
The MITRE Corporation

Abstract

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 paper answers basic questions about software maintenance for a single organization and discusses some of the decisions made based on the answers. Attributes of both the software maintenance process and the resulting product were measured to direct management and engineering attention toward improvement areas, track the improvement over time, and help make choices among alternatives.

Why measure maintenance?

Software maintenance is defined as the process of modifying existing operational software [1]. The importance of software maintenance in today's software industry cannot be overestimated. It is widely recognized as the highest cost phase of the software life cycle with estimated costs of between 60 and 80% of the total software budget [2-4]. It has also been noted [5] that over 50% of programmer effort is dedicated to maintenance. Given this high cost, some organizations are beginning to look at their maintenance processes as areas for competitive advantage [6].

As pointed out by Card, et al., Chapin, and Hariza et al., maintenance of software systems intended for a long operational life pose special management problems [7-9]. Additionally, the software engineering institute propagates the view that organizational processes are a major factor in the predictability and quality of software [10]. Arthur and Stevens explain the importance of documentation to software maintenance [11]. Additionally, Hariza et al, Curtis, and Yuen propose that programmer experience is at least as important as code attributes in determining the complexity associated with software maintenance [9], [12-13]. Thus, the management and planning of large software maintenance organizations must be formalized and quantified.

Our organization became responsible for the maintenance effort on seven products executing in ten locations world-wide in 1994. The seven products contain more than 8.5 million source lines of code written in 22 different languages. Some of the systems have been in the field for more than thirty years while the newest system became operational in 1992. To understand and manage the software maintenance effort we instituted a metrics program based on Basili's Goal/Question/Metric Paradigm [14] and other reported metrics experiences [15-19, 23]. Our goals are to improve customer satisfaction and meet our commitments to cost and schedule. The 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. The results of this exercise are displayed in Table 1. Notice that the questions can support more than one goal. For example, the question "Are we meeting delivery schedules?" supports the goal of customer satisfaction as much as it does the minimize schedules goal. Table 1 shows only one occurrence for brevity.

There are three ways engineers and managers in our organization use the metrics information:

1. Direct attention - what problems do I need to address?
2. Solve problems - which choice should I make?
3. Keep Score - how am I doing?

This paper examines software maintenance in this context. The next section provides an overview of our software maintenance process and describes each step. This is followed by answers to the questions in Table 1 along with process and product changes that are currently being considered. Then we describe some limitations to our metrics program. The summary presents general impressions on software maintenance measurement.

The software maintenance process

Our software maintenance process is depicted in Figure 1. A user identifies something that they do not like (e.g., an improvement or change in requirements) or that does not work properly in a problem report. The problem report is reviewed for completeness by an analyst at the user's location. It is checked against the

1063-6773/96 $5.00 © 1996 IEEE

Table 1. SW Maintenance Goals, Questions, and Metrics

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?	$/delivery
	How are the costs allocated?	$/activity
	What kind 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 duplicate or invalid change requests are evaluated?	% Duplicate and Invalid Change Requests Closed by Month
Minimize Schedule	How difficult is the delivery?	Complexity Assessment
		Software Maintainability
		Computer Resource Utilization
	How many changes are made to the planned delivery content?	% Content Changes by Delivery
	Are we meeting our delivery schedules?	% on-time deliveries

known problem reports to eliminate duplication and is examined to determine if the user may have made a mistake in understanding the system. If it is not a known problem and no user mistake can be identified, the problem report is categorized as a software problem, a hardware problem, or a communications equipment problem. If it is deemed a software problem, a software change form (SCF) is generated. A SCF contains 18 data items including dates for submission and approval, current method, proposed change, justification, and resource estimates.

The SCF is then presented to a user board for validation and to a software maintenance engineer for independent evaluation. The engineer evaluates the SCF based on the effort required to complete the change, the computer resources required, and the quality impact. The resource estimate is made based on the taxonomy of change types described later in the paper. The computer resources and quality impact are made specific to each system. If the independent analysis is significantly different from the user's original estimates a meeting is held to understand and resolve the difference(s).

A SCF can also be generated by another interfacing system or by the maintenance team as a result of technical initiatives and presented to the user board. For example, if System A is upgrading it's communication protocol and it exchanges information with System B, then People from System A can generate an SCF for System B to upgrade its communications protocol.

The user board assigns a categorization to the SCF of either modification or fix. A modification is defined as 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. The priority can be either Emergency, Urgent, or Routine. An Emergency priority is assigned if the SCF is necessary to avoid system downtime or if it is needed to support high-priority mission requirements. An Urgent priority is assigned if the SCF is needed in the next software delivery to support an upcoming mission or fix a problem that arose as a result of a change in operations. A Routine priority is assigned for all other SCFs (e.g., unit conversions, default value changes, fix printout).

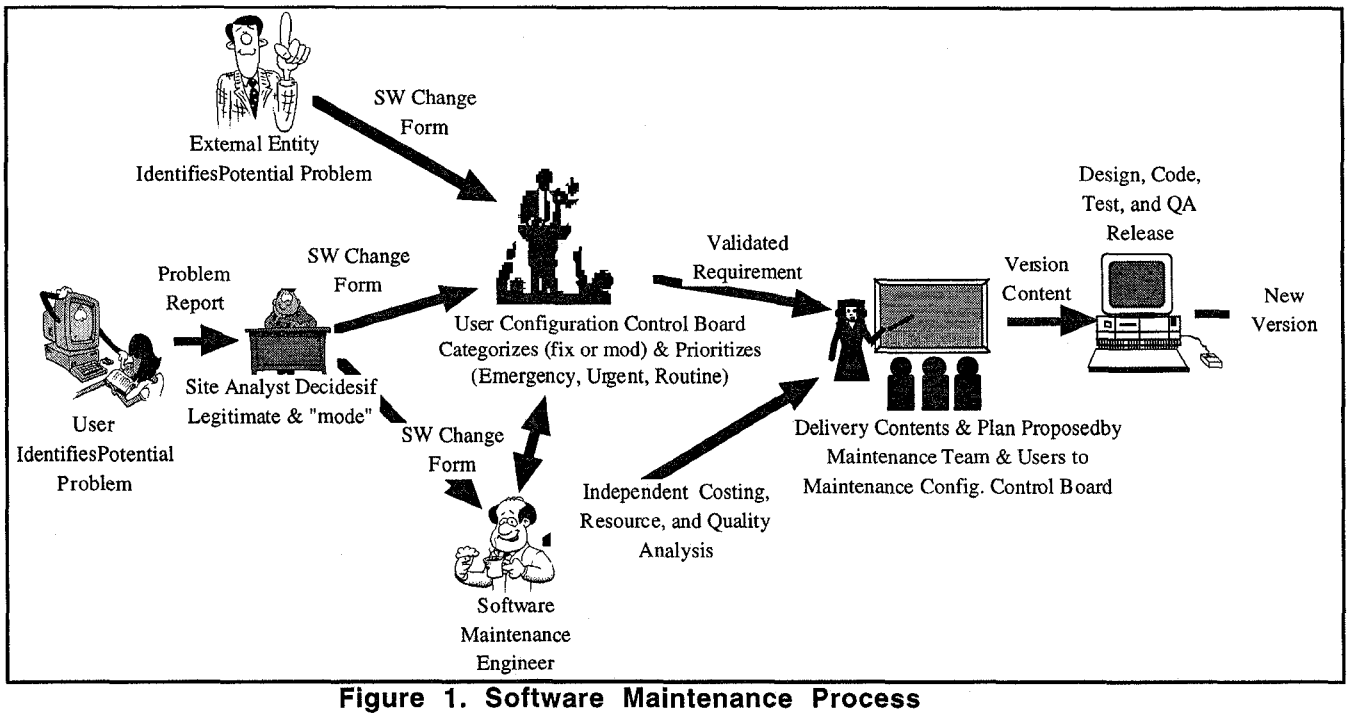

Figure 1. Software Maintenance Process

Once the user board has validated the SCF as a requirement and the maintenance engineer has evaluated the SCF impact, the SCF is placed in a queue for incorporation into a version release. The version release content is negotiated between the users, the maintenance team, and the system engineering organization. Several metrics are examined during this planning and approval process including: release complexity, software reliability, software maintainability, and computer resource utilization. Once the content is agreed to by the maintenance configuration control board, a release is scheduled. The maintenance team then completes the design, coding, testing, installation and quality assurance of the release with user and system engineering review at milestone dates throughout the release.

The metrics in Table 1 are primarily concerned with the four products in Figure 1 that our management team has control over: Requirements Validation, Independent Costing, Delivery Content Definition, and Release Implementation. The earlier phases in the process, while important, have little affect on our organization. Thus, we do not spend resources measuring them.

Maximize customer satisfaction

Through customer surveys and interviews we determined our customer is satisfied with the service they receive if (1) there are not too many problems with the system that affect their ability to complete their job; (2) it does not take too long to resolve complaints the customer has made; and (3) the supplier meets his or her commitments. Four metrics are tracked by our software maintenance organization to address these issues. The following paragraphs discuss each metric.

How many problems affect the customer?

One metric that is tracked over time is the number of customer complaints remaining to solve. Figure 2 shows the backlog of complaints in conjunction with the planned and actual release contents for the past two years for one project. The triangles represent the planned number of changes for a release, the bars are 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 resources (computer and staff), plan release content, and track the effect of new tools or other process improvement programs over time.

For example, in September 1994, it was decided to move the maintenance system from an off-site facility to a consolidated center. This move caused the delay of the planned September release, but since the consolidation every release has been on schedule

because the maintenance team has been able to perform more testing locally and the backlog has dropped. The February 1996 release was an operating system upgrade. Because this release was a single change, the backlog grew during the implementation, future releases are aimed to reduce the backlog.

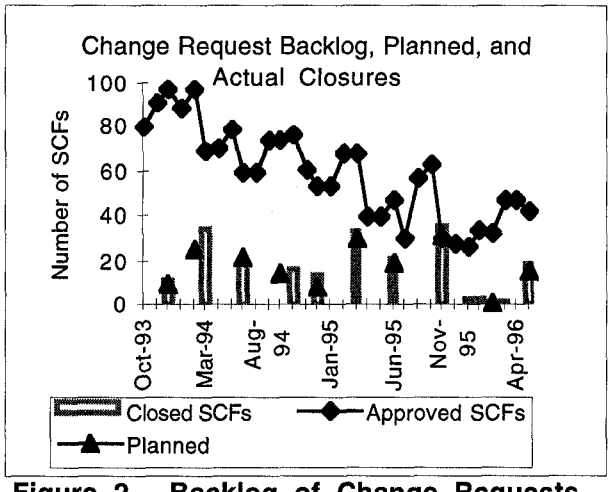

Figure 2. Backlog of Change Requests, Release Plan and Actuals by Month

Obviously, if the backlog remained at zero, a manager would not conclude that no problems are affecting the user. They might conclude that their office is over staffed or in equilibrium with the incoming change requests. An alternative to this metric which looks at the change request arrival rate and closure rate is described in [17].

A second metric that is tracked to understand how many problems are affecting the customer is the software failure rate in the field. A failure is charged to the software if the system does not perform to the user requirements. Each problem report is inspected to determine if it was generated as a result of a software failure. Furthermore, all downtime incidents (which are 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 1000 operational hours to less than 2.4 events per 1000 hours of operation. This improvement in product quality shows up in the backlog chart of Figure 2 since fewer change requests are generated when the system operates without failure for longer periods of time.

This information is used by the system manager in several ways. First, a threshold has been set at 4 failures/1000 hours of operation for this system. If the system is operating well-below this threshold, the manager may decide to incorporate more difficult changes into the next release, whereas, 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, the graph can be used to estimate the number of events during a mission as well as the probability of completing a mission of duration x 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% 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. If the customer requests a major system upgrade, the historical failure rate can be used as a quality requirement to trade-off the cost and schedule against [20].

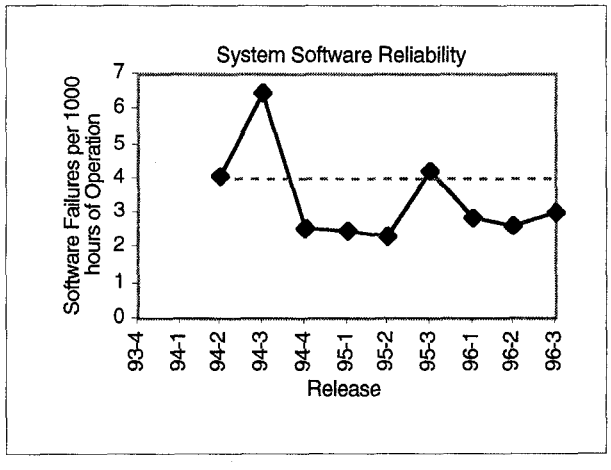

Figure 3. Software Failure Rate by Delivery

How long does it take to fix a Priority Problem?

This is another question related to customer satisfaction. Figure 4 is used to answer the question. It is a cumulative distribution graph of two important process duration. The line with the square markers represents the percentage of priority changes delivered within a given number of days from the time the change was written by the user. The line using the diamond markers shows the time from when the requirement was validated by the user board. The

horizontal distance between the two lines on the chart is the "in process" time. That is, the time the change spent in board review. For this system, the in-process time averaged 54 days. The chart also shows that approximately 80% of all urgent change requests are installed and ready for use by the customer within 90 days of their approval by the user board and within 170 days of the time they are written by the customer. Two of the thirty changes closed in FY95 took more than one year to complete with the majority of this time spent in the design, code, test, and installation phase after the user board validated the requirement. This chart directs our attention to the process time and prompts further questions about each step.

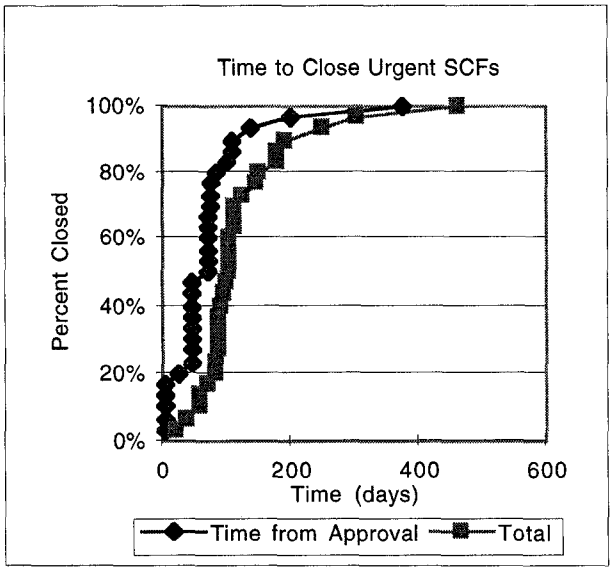

Figure 4. Software Change Form Cycle Times

Thus, these three simple graphs tell us how many problems are affecting the customer, whether or not we are meeting our schedule commitments, and how long it takes to fix a priority problem.

Are we meeting our delivery schedule?

Figure 2 also demonstrates how well the organization is meeting their schedule commitments. It shows that eight of the last ten deliveries were made on schedule. Furthermore, five of the ten had more content than planned while four deliveries had less content than originally planned. The change in content caused the February 1994 delivery to be one month late, while it helped the June 1994 and the June 1995 deliveries to meet their schedules.

Minimize Cost

Many people participate in the software maintenance process. Table 2 lists eleven cost activities for a software maintenance cycle. These costs are further broken into categories of common and system specific costs. The Software Development Activities, Hardware System Maintenance, Travel, and Project Management Categories are system specific costs. Hardware maintenance is a system specific cost since each of our systems use different hardware platforms requiring different levels of hardware maintenance. Travel is apportioned to a release rather than an activity simply to ease accounting.

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. The common costs are allocated to each release as a cost/day.

Table 2. Cost Categories in Software Maintenance

Software Development Activities	Configuration Management	Quality Assurance
Security	Administrative Support	Travel
Project Mgt	System Engineering	HW System Maintenance
System Mgt		Finance

How much does a delivery cost?

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

Deliveries three and ten were the most expensive with the majority of this cost being driven by the difficulty of the environment and the lack of an enforced process. One key to cost improvement and control is understanding the variability in these release costs. Our organization is working to identify the key cost drivers using our delivery complexity model that will be described later in this paper.

How are the costs allocated?

The Pie chart in Figure 6 shows the distribution of costs by activity for a typical release. Based on this chart the cost associated with the Sw Development

Activities (design, code, unit test, and integration test) and hardware system maintenance make up 88% of the total release cost and are being examined to reduce the delivery cost.

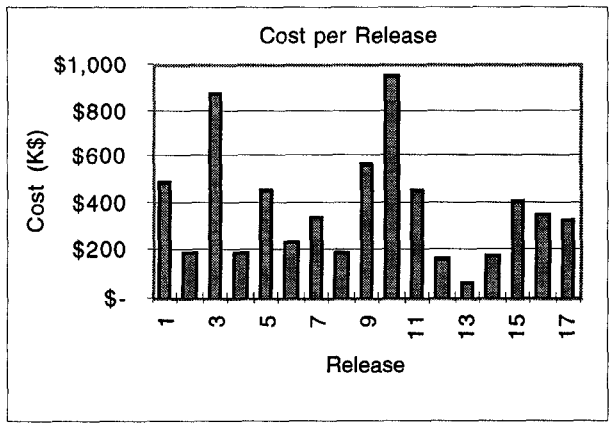

Figure 5. Cost per Delivery for One Year

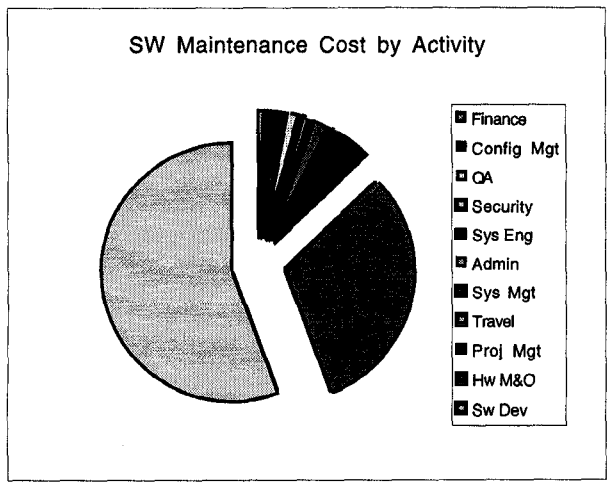

Figure 6. SW Maintenance Cost by Activity

What kind of changes are being made?

The next question about the cost of a release is "What kind of changes are being made?". To answer this question a software change taxonomy was developed using historical change data. The taxonomy is shown in Table 3 and includes ten types of changes and several root causes for each change type.

The changes implemented in the last eight deliveries were categorized using this taxonomy. During the last eight deliveries 67 modification (38%) and 110 fix (62%) changes were made. Figure 7 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 percent frequency of defects and is a convenient scale from which to read the line graph. The line graph connects the cumulative percents (and counts) at each category.

Table 3. Software Change Taxonomy

Change Type	Root Cause
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 SW 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

Figure 7 indicates that Logic changes are the most common (37 changes or 20% of the total) to the software (Although not shown in Figure 7, the majority root cause is missing logic or condition tests for error handling). Using this information we have expanded our design and code reviews to specifically look for these logic problems. Only 1 of the 177 changes analyzed involved data input problems.

How much effort is expended per change type?

Figure 8 is a Pareto diagram of the effort required to make each change. This figure shows that while changes based on requirements or interface specification changes ranked fourth in number of changes with 22, they account for 42% of the total effort at 582 staff-days. Logic changes fall to third when viewed in this manner.

The information from this analysis helps the maintenance engineer make better SCF cost estimates. By reviewing change requests and accurately assigning them to the change taxonomy, he can estimate the staff-days required to design, code, and test individual changes. For example, the average staff-days of effort because of SCFs due to interface specification changes is 36 staff-days with a standard deviation of 43 staff-days, while the average due to requirements specification changes is 22 days with a standard deviation of 24 staff-days. This data is then combined with the distribution from Figure 6 to allocate costs to each of the activities. The actuals are then tracked against the estimate and the information is updated as each release is completed.

While the data is currently highly variable for each root cause, as more data is collected we expect the actual efforts to converge around a reasonable mean to increase our confidence in the estimates. Sudden changes could indicate a need to reexamine our processes or a need to change the staff implementing the SCF. Even with the current variability, we believe using the historical data is the best method for estimating individual change effort.

How complex is the delivery?

We agree with Curtis, when he says complexity is "a loosely defined term" [21]. It is usually concerned with understanding the structure of the code itself and several measures with this goal in mind have been proposed [22-24]. 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.

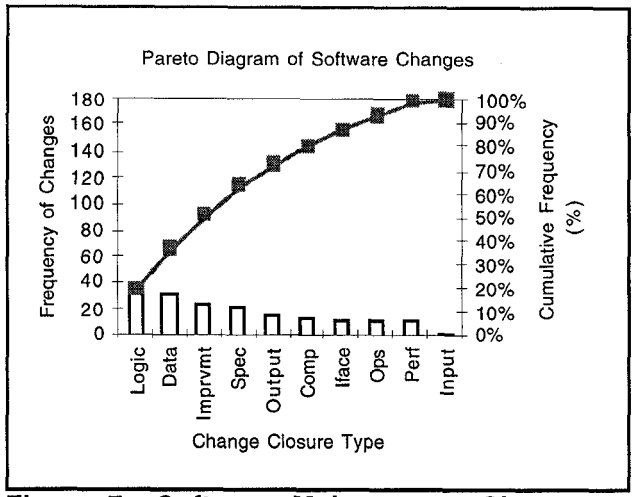

Figure 7 Software Maintenance Changes by Type

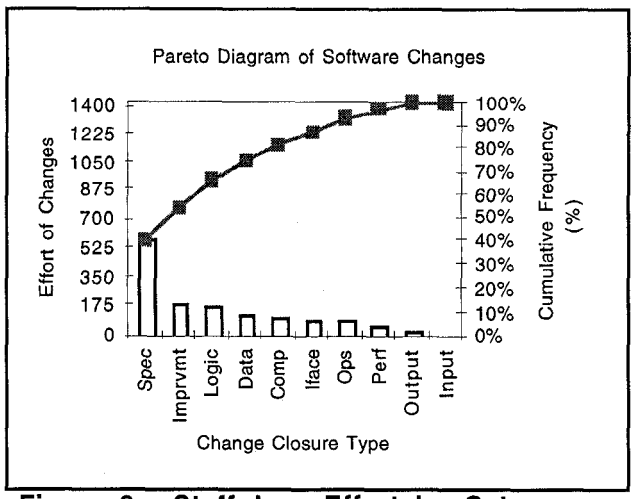

Figure 8. Staff-days Effort by Category

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

After a set of changes is proposed for a delivery, we evaluate the complexity of the proposal using the spreadsheet-based tool described in [25]. The tool calculates a release complexity between 0 and 1 based on a set of objective and subjective data supplied by the system engineer. After using the tool for one year, we found that we are more likely to meet our cost, schedule, and quality targets if the release complexity can be managed below a 0.5 value. Thus, for each release proposal, the results of the tool are evaluated

and, if necessary, complexity reducing techniques (e.g., less content, more experience staff, more rigorous process, better tools, etc.) are applied to the release.

For example, Figure 9 shows the final results of a delivery complexity analysis. In this case, the first use of the complexity tool helped the manager recognize that this delivery was complex in terms of the product. It had a large number of changes compared to previous releases, impacted a lot of modules in the system, and made a lot of external interface changes. The changes involved many different languages and consisted of manipulating the real-time exception processing. The ease and completeness of test diagnostics was a concern as was the difficulty of installation for the release. To mitigate this product complexity the manager assigned his best team to the release. During the second iteration, he found that the overall release complexity was still above the 0.5 rule-of-thumb and that the process and environment were driving factors since the goal was to include all of 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 these decisions caused the documentation to be the major complexity driver. The overall complexity, however, was reduced to a typical level for this project and the delivery was implemented. The project was delivered on schedule, less than 5% over budget, and with no reported defects during user trial period.

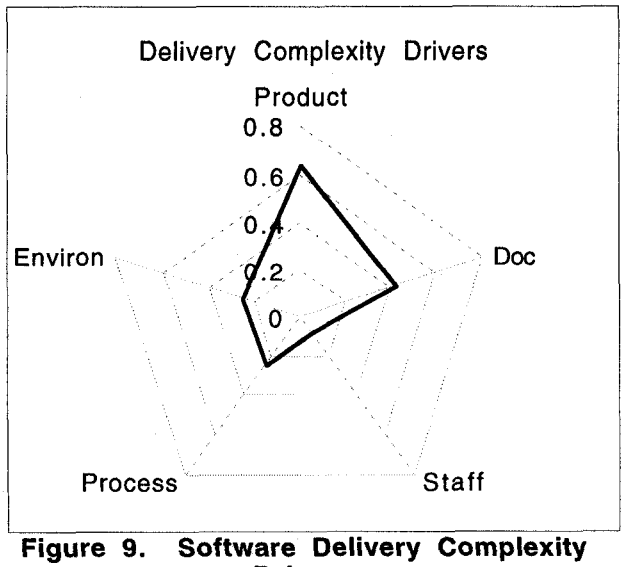

Figure 9. Software Delivery Complexity Drivers

How many duplicate and invalid change requests are evaluated?

Throughout the software change process, a problem report or SCF may be closed as an invalid or duplicate request. Even after a SCF is approved as a valid requirement it may be closed based on another change that has fixed the problem or some external event (for example, site closing or external interface change). The percent of invalid and duplicate change requests that are closed out of the total evaluated is a measure of rework in our process. Figure 10 shows the data for 6 quarters. On the average 72 change requests are evaluated each quarter. On the average 8% of the total change requests evaluated each quarter are duplicates or invalid. This is not a large percentage and translates to only $7500 per year (5.5 hrs/eval x 5.75 rework evals/qtr * 60 $/hr * 4 qtr/yr).

While it is not a significant cost to our organization, we are still interested in approaches to reducing the amount of rework in our process. Two low-cost initiatives that we have targeted are to improve user training on the system (regularly schedule sessions and update material) and to share the active backlog list with all personnel involved in the process.

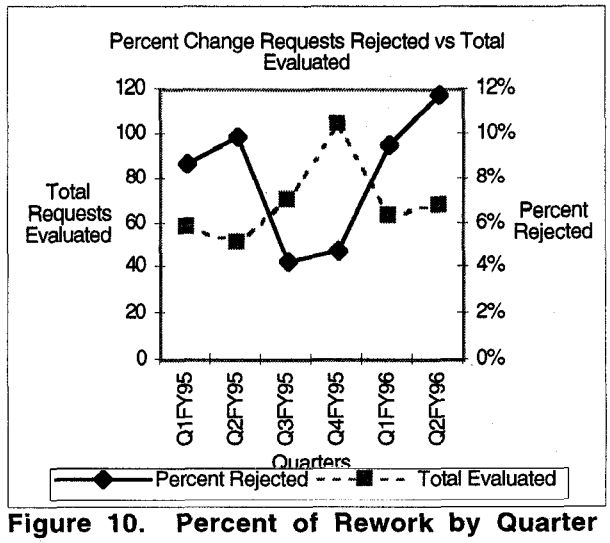

Figure 10. Percent of Rework by Quarter

How many changes are made to the planned delivery content?

Once a delivery plan is agreed upon by the customer, the developer, and the system manager a major factor in performing to the plan is requirements volatility. Requirements volatility comes in three types, additions to the delivery content, deletions from the delivery content, and changes to an agreed to request. The requirements volatility for 19 deliveries is in Figure 11.

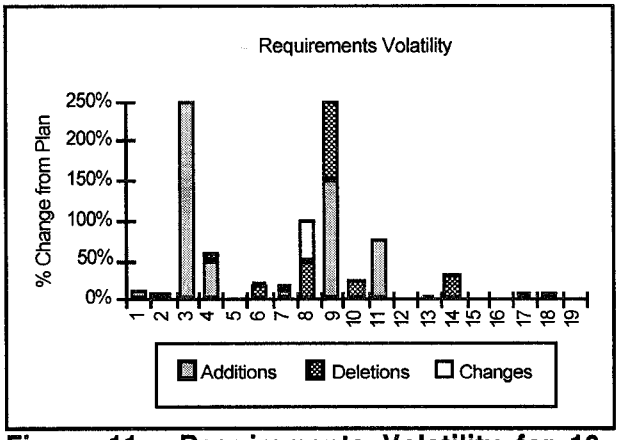

Figure 11. Requirements Volatility for 19 Deliveries

Five of the nineteen deliveries (26%) had no requirements change; of these, two were made ahead of schedule and all were within 15% of the originally scheduled delivery date. Fourteen of the nineteen (74%) had requirements change with five of them having greater than 50% change. Because requirements volatility is a part of our process, we looked for a method to quantify its impact.

A scatterplot of schedule performance vs. requirements volatility for these nineteen deliveries is shown in Figure 12. A simple linear curve fit (least squares) to predict schedule volatility was developed using this data and the delivery effort data from Figure 8. The model is of the form:

Percent Schedule Slip = $0.96 + 0.41*(\%\ \text{rqmt vol})^{1/2} + 0.23*(\text{changes/staff-day})$

This equation yields a coefficient of correlation (R^2) of 0.72 with a standard error of 0.17. Notice that the schedule change goes up regardless of whether the requirements change was an addition or deletion because the input to the model is percent of requirements 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, while others argue that a reduction in requirements means less work for the team and should result in completing the project in less calendar time.

The equation is used to explain 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; the customer wanted to drop two of the requirements and change the scope of a third at preliminary design. Managers estimated the change in risk to version delivery to change from 0.14 (15 changes in 108 staff-days) to 0.1 (13 changes in 130 staff-days). Using the model, managers forecasted the overall schedule impact to be $[0.97 + 0.41*(0.2)^{1/2} + 0.23*(0.1)] = 1.18$ or an 18% schedule slip. An 18% slip is equivalent to 16 days added on to the 91 day schedule.

During discussion about the model and the prediction, the customer decided that this slip was not acceptable to the overall mission of the version. Thus, they decided not to pursue the changes, but to incorporate the scope change in the next release. The metrics-based model facilitated objective communication about version release plans and status between us and our customer.

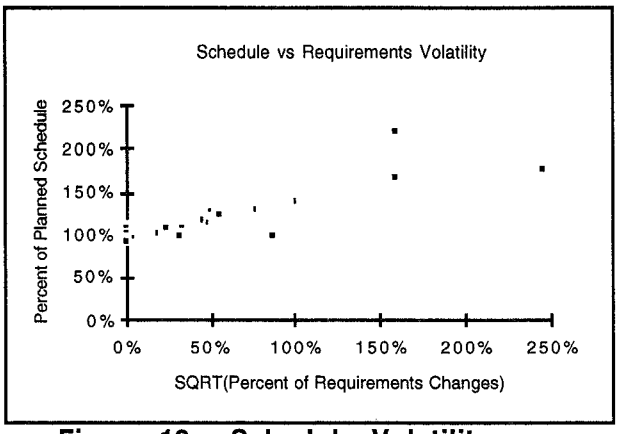

Figure 12. Schedule Volatility vs. Requirements Volatility

Limitations

While our measurement program is providing benefits, it also has three major limitations. First, we do not have an acceptable way to measure and manage software maintainability. We have tried to quantify the maintainability of our software and the impact of releases on maintainability using the survey-based approach proposed by AFOTEC [26] and the polynomial approach proposed by Oman, et al [27]. Neither of these approaches have proven repeatable across our systems.

Second, we do not have a method for predicting 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% busy, memory below 98% 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 comparable way to other programs. We measure size in terms of the number of changes completed rather than source lines of code changed or function points delivered because of early confusion caused by the two

more common metrics. This makes it difficult to compare our productivity and process throughput with other organizations or "industry averages."

Summary

Our software maintenance metrics program is only two years old. In this short time, however, we have reaped many benefits from measuring our products and our processes. We believe measurements enhance software maintenance management. We began with clearly defined goals targeted to project managers and engineers. Questions were asked to assess the current state of the product and process and 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 paper describes several decisions and examines the limitations of our program. Hopefully, 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.

References

[1] B. Boehm, Software Engineering Economics, Prentice-Hall, New York, 1981.

[2] B. P. Lientz and E. B. Swanson, "Characteristics of Application Software Maintenance," *Comm of ACM*, vol. 21, no. 6, Jun. 1978, pp. 466-471.

[3] G. Parikh, The Guide to Software Maintenance, Winthrop Publishers, Cambridge, Mass, 1982.

[4] S. S. Yau and T. J. Tsai, "A Survey of Software Design Techniques," *IEEE Trans on SW Eng*, vol. 12, no. 6, Jun. 1986, pp. 713-721.

[5] V. Gibson and J. Senn, "System Structure and Software Maintenance Performance," *Comm of ACM*, vol. 27, no. 3, Mar. 1989, pp. 347-358.

[6] J. Moad, "Maintaining the Competitive Edge," *Datamation*, vol. 36, no. 4, Feb. 1990, pp. 61-66.

[7] D. N. Card, D. V. Cotnoir, and C. E. Goorevich, "Managing SW Maintenance Cost and Quality," *Proc. Intl. Conf on SW Maint.*, Sept. 1987.

[8] N. Chapin, "The Software Maintenance Life-Cycle," *Proc. Intl. Conf on SW Maint.*, 1988.

[9] M. Hariza, J. F., Voidrot, E. Minor, L. Pofelski, and S. Blazy, "Software Maintenance: An Analysis of Industrial Needs and Constraints," *Proc. Intl. Conf on SW Maint.*, Orlando, Fl., 1992.

[10] Software Engineering Institute (SEI), "Software Process Maturity Questionnaire Capability Maturity Model, version 1.1," Carnegie Mellon Univ., Pittsburgh, PA, 1994.

[11] J. Arthur and K. Stevens, "Assessing the adequacy of documentation through document quality indicators," *Proc. Intl. Conf on SW Maint.*, 1989.

[12] B. Curtis, "Conceptual Issues in Software Metrics," *Proc. IEEE Intl. Conf. on System Sciences*, 1986.

[13] C. Yuen, "An empirical Approach to the Study of Errors in Large Software Under Maintenance," *Proc. Intl. Conf on SW Maint.*, 1985, pp. 96-105.

[14] V. Basili, and H. D. Rombach, "Tailoring the software process to goals and environments," *Proc. 9th Intl. Conf on SW Eng.*, Monterey, CA, 1987.

[15] R. B. Grady, "Measuring and Managing Software Maintenance," *IEEE Software*, vol 4., 1987.

[16] G. E. Stark, R. C. Durst, and C. W. Vowell, "Using Metrics for Management Decision-making," *IEEE Computer*, Sept. 1994.

[17] G. E. Stark, L. C. Kern, and C. W. Vowell, "A Software Metric Set for Program Maintenance Management," *Journal of Systems and Software*, vol. 24, pp. 239-249, 1994.

[18] R. B. Grady, Practical Software Metrics for Project Management and Process Improvement, Prentice-Hall, Englewood-Cliffs, NJ, 1992.

[19] T. Lydon, M. Wall, and L. Fischer, "Software Metrics at Raytheon," *Proc 4th Annual Oregon Workshop on Software Metrics*, March, 1992, Silver Falls, OR.

[20] J. D. Musa, A. Iannino, and K. Okumoto, Software Reliability: Measurement, Prediction, and Application, McGraw-Hill, New York, 1987.

[21] B. Curtis, "Measurement and experimentation in Software Engineering," *Proc. of IEEE*, vol 68, no. 9, 1980, pp. 1144-1157.

[22] T. McCabe, "A Complexity Measure," *IEEE Trans. on SW Eng.*, SE-2, 1976, pp. 308-320.

[23] N. Fenton, Software Metrics: A Rigorous Approach, Chapman and Hall, London, 1991.

[24] D. Kafura and G. R. Reddy, "The Use of Software Complexity in Software Maintenance," *IEEE Trans SW Eng*, vol SE-13, no. 3, 1987.

[25] G. E. Stark, and P. W. Oman, "A Survey Instrument for Understanding the Complexity of Software Maintenance," *Software Maintenance: Research and Practice*, vol 7, Dec. 1995, pp. 421-441.

[26] AFOTEC, Software Maintainability–Evaluation Guide, AFOTEC Pamphlet 800-2, vol 3, HQ Air Force Operational Test and Evaluation Center, Kirtland Air Force Base, New Mexico 87117-7001.

[27] P. W. Oman, D. Ash, J. Alderete, and B. Lowther, "Using software maintainability models to track code health," *Proc. Intl. Conf. on SW Maint.*, 1994, pp. 154-160.