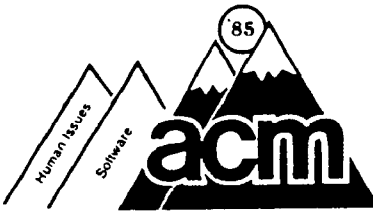

SOFTWARE PRODUCTIVITY AND QUALITY MEASUREMENT

Lowell Jay Arthur
Mountain Bell
Denver, Colorado

Methods of measuring programmer productivity and software quality will be presented, including the author's experience at Mountain Bell. Static code and documentation analyzers, change management systems, and other automated measurement tools will be discussed in their relationship to software development and maintenance.

Information Systems (IS) managers often overlook quality improvements as a means to achieve productivity improvements. The timeless adages "There's never time to do it right, but there's always time to do it over" and "Quality is free" overstate what should be intuitively obvious. High quality products require less maintenance. They enhance both development and maintenance productivity. Software quality measurement can achieve these goals.

In the average company, system maintenance and enhancement consume 50-80% of the IS budget. Why is that? In most cases, the programs were poorly written to begin with. Then, they were enhanced in ways that caused them to decay rather than improve. And finally, the programs became such a burden that management had to invest in rewriting them. Unfortunately, the next version of the system was not written any better than its predecessor. This recurring failure stems from the current inability to measure and improve quality.

What is Software Quality?

Software quality is composed of many factors: maintainability, flexibility, reliability, reusability, efficiency, and a host of other quality related metrics. Each of these factors can be measured at some time during the software life cycle. Only when measurement is applied routinely do the benefits become visible.

What are the benefits of measurement?

Statistical quality control (SQC) has been widely used in manufacturing to reduce costs. Quality control, based on measurement, helps reduce costs by eliminating defects in developing systems and by identifying maintenance problems. Costs are also reduced because programmers and analysts eliminate scrap (throwing away bad code), rework (fixing defective products), and downtime (productivity losses).

Software measurement provides benefits to managers, system developers, and maintenance personnel. Management benefits from early identification and resolution of problem areas. Because their software products will be more reliable and maintainable, clients will perceive IS management as more responsive. Finally, investment in quality control and improvement typically maximizes the corporation's return on investment:

20% of the programs cause 80% of the costs

1985 ACM 0-89791-170-9/85/1000-0187 $00.75

By focusing on the programs that consume the majority of the IS budget, IS managers can work to eliminate the costs associated with maintaining these programs. Management can even identify poor quality in developing programs and take corrective action before the programs are compiled or tested. Coding is only 10% of the development process. Quality changes can be effected with little impact on the development schedule. If anything, quality changes will shorten the development process by reducing testing costs.

In a development environment, software quality measurement can provide programmers with immediate feedback about the quality of their emerging code. They can take corrective action without a walkthrough. Improvements in code quality will ultimately be reflected in the maintenance costs for the new system. Programmers will have more time to enhance the existing system or to develop new ones, and this means greater productivity.

In a maintenance environment, software quality measurement helps identify reliability problems, improve estimation techniques, and improve maintainability. Tracking the kinds and frequency of program errors helps identify reliability problems. Knowing the size and complexity of existing code helps estimate maintenance costs. Software metrics can help provide the ammunition to convince management and clients to invest in correcting poor code. You wouldn't expect a car to run forever without an oil change or a computer to operate without preventive maintenance. Why would you expect anything different from your software? Software metrics can identify quality problems and suggest solutions that can be integrated into the normal maintenance process. To understand how all of these things are possible, you must first understand the basics of quality measurement.

How is Quality Measured?

Software quality measurment applies to all stages of the software development life cycle. Software quality measurement can occur in any phase of development and maintenance using manual design inspections, prototyping, static analysis, dynamic analysis, operational analysis, and change management tracking.

Design inspections, although human intensive, help achieve quality in system and program designs. Prototyping provides immediate feedback about the system's usability. Software metrics, based on a mechanized analysis of the system's code, provide a means to quantify many important quality characteristics before a module is compiled or tested. Dynamic analysis helps identify efficiency problems. Operational analysis measures reliability. Change management tracking helps determine maintainability, flexibility, and reliability. Unlike other methodologies, software quality measurement can be applied to both the development and maintenance of code.

Static Analysis

Software quality measurement should be applied as soon as possible in a developing system to identify and eliminate quality problems. Since measurement during the design phase is largely manual, you should look to the coding phase as the first opportunity to mechanize quality measurement. Static analysis, in the form of code analyzers, can take many quality measurements. Since code is the major product of the development process, these software metrics analyzers form a major defense against quality problems. Because code analysis is easily automated, static analysis has received the lion's share of study and verification--another reason for relying on static analysis. It can be used in both the development and the maintenance of software.

Static analyzers predict code quality using two forms of measurement: size and complexity. Size measurement using executable lines of code (ELOC), functions, and software science metrics has been widely correlated to development and maintenance costs. Complexity measurement, using decision-based

metrics, has also been widely validated and correlated with productivity. Both size and complexity measurements give managers, analysts, and programmers information to aid in quality control.

The size of a module can vary from a few executable lines to many thousand. Yourdon has recommended modules no larger than 50 ELOC. Barry Boehm found that 100 was equally reasonable. I have found that for a module to possess Glenford Myers' two reliability criteria--data coupling and functional strength--it will almost always fall within 100 ELOC. There are only two exceptions: large CASE constructs that evaluate hundreds of possibilities (for example, item numbers) or modules that edit transactions with hundreds of fields. At the code level, no better, simple metric of size exists than ELOC.

Functions can be determined from the number of paragraphs in a COBOL module or PROCEDURE statements in a PL/1 module. Functions have been correlated with productivity (Crossman 1979). Alan Albrecht's function point metrics based on inputs, outputs, inquiries, master files, and external interfaces, have gained wide acceptance for estimating system complexity and developmental costs. Both of these size metrics can be applied at a module level, but are normally used at the system level.

Software science metrics are a more complex set of size measurements that depend on both the executable lines of code and the number of data items used in a module. Based on these elementary metrics, Halstead suggested many quality related metrics: Length, Volume, Difficulty, and Effort. Volume and Effort have been widely correlated with program defects, quality, and productivity. An IBM study, however, found no significant difference between ELOC and the software science metrics as measures of size.

Decisions--IF-THEN-ELSE, CASE, DOWHILE, and DOUNTIL--are the basis of most complexity measures. Intuitively, this seems reasonable because each decision adds two or more test paths to the code. The complexity of the logic depends on the number of paths through the code. Reliability, maintainability, flexibility, and testability have all been correlated to decision complexity. A simple count of the number of decisions in the code gives a basic metric of complexity. A module of 100 ELOC can have complex decision logic without affecting its quality; people can understand two pages of code. In modules larger than 100 ELOC, however, productivity and quality suffer as the number of decisions increase.

A widely validated extension to decision metrics, called Cyclomatic Complexity, was introduced by Thomas McCabe. It is based on graph theory, but boils down to the addition of logical operators (AND, OR, and NOT) to the count of decisions. Each logical operator is a thinly veiled IF statement and must be included to truely reflect complexity. McCabe found that modules possessing a Cyclomatic Complexity of 10 or less contained no errors. Most modules under 100 ELOC contain 10 or fewer decisions.

The use of GOTOs (a decision with only one outcome) also increases complexity. McCabe found that there were only four kinds of structure violations: branches into or out of a decision or loop. What has to exist to allow these violations? The GOTO. GOTOs also make it difficult to reduce the complexity of a program by spliting it into functional modules.

Size and complexity metrics like ELOC and Cyclomatic Complexity are easily obtainable from code. Since coding is one of the earliest phases in which to detect and correct quality problems to minimize future costs, static analyzers should be implemented to improve code quality.

Dynamic Analysis

During testing, dynamic analysis helps identify efficiency problems. If a program runs for two hours in testing, you know it will run longer in production. Efficiency improvements and checkpoint restart capabilities can be added at this time.

Operational Analysis

Reliability problems will crop up first in testing. Program A will fail several times compared to program B that never fails. Management should invest in reliability analysis and correction of program A. In testing as well as in production, analysis of operational logs can provide reliability metrics like mean time between failures (MTBF) and mean time to repair (MTTR). Operational analysis can be mechanized via a variety of products that evaluate operating system logs.

Change Management Tracking

Across all of the phases of development and maintenance, problems and enhancements will be identified. Tracking these requested changes gives insight into the reasons for quality problems and offers an opportunity to prevent the same problems in newly developed systems. Change management tracking also identifies systems, programs, and modules that require the most changes. This data can then be used in combination with actual work effort to propose quality improvements that will reduce maintenance costs. Problem tracking systems are few, but they will gain acceptance as software engineers observe, like their manufacturing counterparts, that quality tracking is the most important element in developing a quality improvement plan. Without historical data, Information Systems personnel are doomed to repeat mistakes that could otherwise be avoided.

How do you use software quality measurement?

As we have seen in the past, the development of each new system adds to the burden of maintenance. The maintenance of existing systems becomes more difficult with each enhancement or program fix. To maximize the company's return on investment, each new product must be built to minimize maintenance costs, program failures, and run time. These new systems need to be designed to maximize maintainability, flexibility, reliability, and efficiency. To improve productivity, new systems will have to take advantage of reusable code. Security will be an issue. Programmers can code toward specific quality goals. Coding to maximize productivity only serves to minimize software quality and maximize long-term costs.

Development

Using the qualities previously discussed, you should establish early in the development process the qualities that are most desireable in the finished product. Analysts and programmers can meet those goals.

One quality that is crucial to productivity is reusability. On one software project I worked on at Bell Labs, the first version of the system relied on extensive use of reusable code and modules. Maintainability and reusability were the software quality goals of the development team. The development team delivered 70,000 ELOC. Expanding the system to account for all occurrences of reusable code, the delivered system contained 260,000 ELOC. It would have taken four times as many people to develop and maintain the system without reusability.

Once software quality goals like these have been established, you should use as many of the forms of measurement as possible to track the development of quality in the emerging system. Take corrective action whenever necessary. Do not allow quality problems to slip into the finished product.

Maintenance

There are three types of maintenance: repair, enhancement, and preventive. Most Information Systems organizations practice repair and enhancement maintenance; few practice preventive maintenance. Repair and enhancement maintenance has to get done. Preventive maintenance happens when there is time. There is never time. Yet preventive maintenance offers the maximum potential for productivity and quality improvement.

Maintenance consumes 80% of the typical IS budget. Yet there seems to be a reluctance or an inability to find ways to reduce that burden. Software quality measurement will help identify the 20% of the programs in maintenance that incur 80% of the costs. These programs should be modified, upgraded, or rewritten to reduce maintenance costs. Rewriting programs comes under the heading of new development--quality goals should be selected, measured, and achieved. Modifications and upgrades are the province of preventive maintenance.

Once a program or module has been identified as a candidate for preventive maintenance, an editor (not the programmer) should be chosen to review and revise the code. In programs where size is a problem, the editor should look for and eliminate redundant code. In a typical program over 100 ELOC in length, 10-20% of the program code is redundant. In the case of one program that was over 2200 ELOC, I was able to remove 700 ELOC, insert references to the reusable code, and test the program in a single day. Based on current research, this means that I permanently reduced the future maintenance costs by 30%.

Once an editor has removed the duplicate code, he or she should then attempt to reduce decision complexity. Eliminating NOT logic is one of the simplest ways to reduce decision complexity--Cyclomatic Complexity decreases every time a NOT is removed. Next, the editor should look for any way to restructure the logic to reduce the number of decisions and decision nesting. Eliminating all GO TOs will further reduce complexity because there can be no structure violations.

At this point, the module has been reduced to its simplest form. If the module conforms with basic quality goals, it can be tested and released. If not, it should be broken down into several functional pieces that meet the quality goals. Most programs contain a significant number of reusable functions. A payroll program might contain a Social Security number edit that is reusable. Another module could contain report headers, footers, and data formatters that could be separated. The editor should make these further alterations, although they may take several days. In the end, the program will be maintainable, flexible, and more reliable. Because reusable modules have been isolated, future development efforts can use them to improve productivity. Maintenance costs will shrink to a fraction of their previous levels because there is less code to maintain and each module is less complex. These benefits are available for the investment of 1-5 days per module. Compared to the cost of rewriting such a program (possibly several weeks), preventive maintenance offers a way to ease out from under the IS maintenance burden, releasing programmers and analysts to tackle the system development backlog.

Productivity Measurement

Traditionally, Information Systems managers have measured productivity by means of subjective performance evaluations. Performance evaluations aim at understanding the physical, technical, and economic phenomena that contribute to changes in productivity, costs, and profits. These measures only indicate what an Information Systems department has done, not what it is capable of doing. How much is enough, when looking a productivity? How is productivity measured?

Productivity measurement most often refers to the relationship between inputs and outputs, typically a ratio of outputs to a single input such as

output per person day, or in the IS case, lines of code (LOC) per person day. These ratios tend to lead to misinterpretations; increases in output may or may not be desirable. Elementary productivity theory indicates that productivity changes occur through changes in the quantity and quality of the inputs. For example, reducing responste time to two seconds may optimize a programmer's productivity at the terminal. The use of non-procedural languages or office automation systems can also affect productivity. Training IS professionals in the latest methodologies (adjusting the quality of the people resource) can also increase productivity. From these simple examples, it should be apparent that the analysis of productivity has to recognize the interaction of all inputs, integrating them into a set of meaningful measurements.

The essential inputs to the development and maintenance process are:

People - days worked
Machines - CPU hours
Costs - Capital and Expense

The essential outputs to be measured are summaries of the quality metrics already available:

Executable Lines of Code (ELOC)
Functions
Pages of Documentation
Software Science Metrics

Productivity metrics are then nothing more than ratios of each output to each input: output/input. Rather than depend on any one productivity metric, which tends to cause misinterpretation and all kinds of problems, managers should examine the entire network of productivity ratios and determine what they mean as a whole rather than individually.

This examination simply tells whether the IS department is on the right track, if new technology is working, if new methodology is improving quality, and so on. Ratios like functions/person day, ELOC/$, and effort/CPU hour each paint a different picture of the productivity of IS. Management should examine each metric's validity in their work environment. Don't say "It doesn't work!" Say instead that "I can't use it in my work area," because the manager next to you may find the metric to be most helpful.

Remember that:

1. Productivity analysis serves a variety of masters, requiring a corresponding variety of carefully designed metrics.
2. The productivity of any system should refer to an integrated network of output/input ratios. No single measure can depict accurately all of the vagaries of productivity.
3. Productivity adjustments depend on changes in the output/input ratios and managerial choices for harnessing the increased productivity.
4. The network of output/input ratios must be supplemented with cost measures and other criteria until they correctly reflect the objectives of productivity measurement.

Future Directions

Each IS organization has a large base of existing code that eats a significant portion of the budget. Remaining resources are used for new development using conventional technology. Organizations that are just beginning to use fourth generation technology find that productivity and quality decline during the first few projects. You will need measurements to determine the benefits of new technology and methodology. Quality assurance groups are needed to develop, maintain, and collect productivity quality measurements.

Productivity and quality measurements are tools to aid in the development and maintenance of high quality software. They not only stimulate better quality, but also higher productivity. As a result, software measurement serves as a basis to elevate software development one more notch toward the goal of software engineering.