Chapter 1

Why Measure?

The papers

Measuring for understanding

Norman Fenton, Shari Lawrence Pfleeger, and Robert L. Glass, "Science and Substance: A Challenge to Software Engineers," *IEEE Software*, Vol. 11, No. 4, July 1994, pp. 86–95. Describes how measurement is key to understanding and evaluating the factors that affect our products, processes, and resources.

Measuring for experimentation

Victor R. Basili, Richard W. Selby, and David H. Hutchens, "Experimentation in Software Engineering," *IEEE Trans. Software Eng.*, Vol. SE-12, No. 7, July 1986, pp. 733–743. Presents a framework for analyzing experimentation in software engineering to help structure the experimental process and provide a means of classifying previous work.

Measuring for project control

Barry W. Boehm, "Software Risk Management: Principles and Practices," *IEEE Software*, Vol. 8, No. 1, Jan. 1991, pp. 32–41. Uses four subsets of risk management techniques to identify risk items and rate a project's current status, ranking risk items by their risk exposure values.

Measuring for process improvement

Michael K. Daskalantonakis, "Achieving Higher SEI Levels," *IEEE Software*, Vol. 11, No. 4, July 1994, pp. 17–24. Describes how to conduct and monitor incremental assessments when improved SEI capability maturity level is the long-term goal.

Measuring for product improvement

Robert B. Grady, "Successfully Applying Software Metrics," *Computer*, Vol. 27, No. 9, Sept. 1994, pp. 18–25. Uses examples from real projects to show that a project's success depends on using clearly defined measures to aid design and management decisions.

Measuring for prediction

Edward F. Weller, "Using Metrics to Manage Software Projects," *Computer*, Vol. 27, No. 9, Sept. 1994, pp. 27–33. Shows how defect data collected over time can be used to plan projects and schedule delivery, with an overall improvement in software process maturity.

Case study

George Stark, Robert C. Durst, and C.W. Vowell, "Using Metrics in Management Decision Making," *Computer*, Vol. 27, No. 9, Sept. 1994, pp. 42–48. Explains how metrics defined using the Goal-Question-Metric paradigm and standardized tool kits help managers at NASA's Mission Operations Directorate to understand better their processes and products.

Editors' introduction

As with any other profession, the quality of our practitioners and processes is judged by the quality of our products. And software's ubiquity means that our reputations are continually on the line. In *The Decline and Fall of the American Programmer* (Prentice Hall, 1991), Ed Yourdon tells us that

> "Today, the world-class software company knows that it cannot be satisfied with what it's doing. [...] Software is now a global industry, and a lot of hungry people around the world are aching to eat your lunch."

To get ahead and stay ahead, a company must deliver products that are better than its competition's. It must strive to improve the status quo. To produce an excellent product, the successful software company must make improvements to every step of the software life cycle. But to do that, each step must be understood. Measurement is critical to understanding; as Lord Kelvin reminded us, "One does not understand what one cannot measure."

We begin our book by examining the reasons for measuring software. There are three major reasons to measure:

- understanding
- predicting
- controlling

As we saw in the foreword, measurement allows us to manipulate symbols in a formal, mathematical way so that we learn more about what is happening in the real, empirical world. Metrics help us to identify unusual cases (good and bad) and to describe a typical or usual situation (that is, to establish a baseline). We can also determine cost, effort, and duration requirements for various types of products or process activities.

By knowing what is typical, we can begin to predict what is likely to happen later in an on-going project, or on future projects. For example, given a metrics database of information about past projects, we can estimate resource requirements on similar projects that may be proposed. Likewise, we can use information about defects already discovered to help us predict the likely number of defects still remaining in our code, and thus the amount of testing required before the product can be delivered. Figure 3 illustrates opportunities for prediction throughout the development process.

Finally, the predictions enable us to control projects. We can estimate the cost, effort, and schedule required to complete a project, and track the actual values against our estimates. By examining information about changes to requirements, design, code, and test cases, we can determine whether the effort and schedule allotments should be revised. We can use defect and test coverage information to set entry and exit criteria for testing and quality assurance activities. And we can use information about unusual cases to determine our testing strategy.

Our measurements offer us control in another sense. Many organizations are concentrating on improving their software products and processes. Metrics information can assist by revealing which activities are resulting in demonstrable improvement. That is, we can use metrics information to help us understand which activities are most effective at accomplishing our goals. For example, Bob Grady compared the efficiency of different kinds of testing techniques, as shown in Table 1. (See [Grady 1992] in this chapter's "To probe further.") His experiment and similar research results confirm that inspections are one of the cheapest and most effective testing techniques for finding faults.

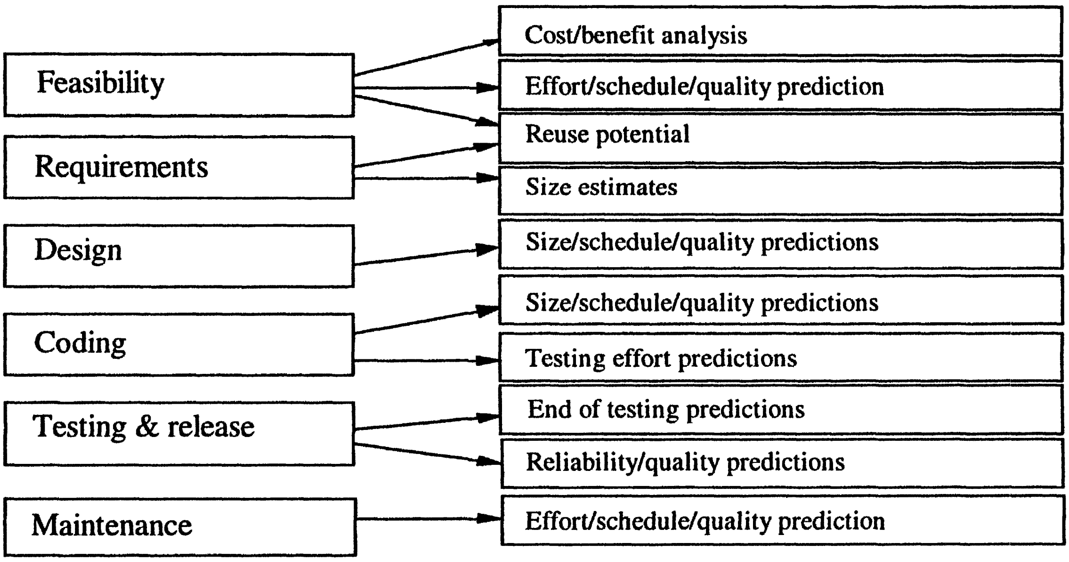

Feasibility
Requirements
Design
Coding
Testing & release
Maintenance
Cost/benefit analysis
Effort/schedule/quality prediction
Reuse potential
Size estimates
Size/schedule/quality predictions
Size/schedule/quality predictions
Testing effort predictions
End of testing predictions
Reliability/quality predictions
Effort/schedule/quality prediction

Figure 3. Example predictions needed for software development decision-making (Fenton and Pfleeger 1996).

Table 1. Comparison of testing efficiency. [Grady 1992]

Testing Type	Efficiency (Defects found per hour)
Regular Use	.210
Black Box	.282
White Box	.322
Reading/Inspections	1.057

The papers in this first chapter focus on the importance of measurement and the rationale for instituting a comprehensive measurement program. In "Science and Substance: A Challenge to Software Engineers," Norman Fenton, Shari Lawrence Pfleeger, and Bob Glass explain that measurement is critical in evaluating the effectiveness of any proposed technology. They explain that careful experimentation is needed to determine the effects of new techniques or tools on the resulting software quality. By presenting guidelines for evaluating experiments and case studies, they help us not only to improve our own evaluative work but also to assess the reported work of others.

In our second paper, "Experimentation in Software Engineering," Victor Basili, Rick Selby, and David Hutchens propose a framework for experimentation in the software environment. The framework is then used to evaluate several reported experimental studies. Thus, the first two papers show us that measurement is simply good engineering practice, and that we cannot be good scientists without it.

Next, we turn to project management to see how measurement can help us to predict and control a project's outcome. Associated with every project are a number of risks, many of which relate to allocation of limited resources. In "Software Risk Management: Principles and Practices," Barry Boehm sets forth a model for managing these risks. Using his model, a development organization can try to reduce the probability of exposure to severe risk. Boehm illustrates his approach with examples, showing that projects using the "concept of risk exposure ... tended to avoid pitfalls and produce good products."

Project management also involves controlling the software process. Successful software managers use experiences on previous projects to help improve the next ones. In "Achieving Higher SEI Levels," Michael Daskalantonakis describes how Motorola conducts and monitors incremental assessments to help projects reach their long-term process improvement goals. Whether you use the Software Engineering Institute's capability maturity levels or some other measure of process quality (such as SPICE, Bootstrap, or ISO-9001), the techniques described here are useful in providing interim feedback toward long-term improvement.

Hewlett-Packard is well-known for its successful corporate measurement program based on H-P's goal of improving product quality. Bob Grady shares his experiences at H-P by instructing us in "Successfully Applying Software Metrics." Grady suggests particular courses of action for metrics managers, using real-life examples to support his arguments. Ed Weller tells us about "Using Metrics to Manage Software Projects" at

Honeywell. Weller's article complements Grady's, showing that metrics collected from past projects can aid in planning future ones.

The first chapter closes with a case study of the successful introduction of metrics at NASA's Johnson Space Center. Realizing that NASA software failures could be catastrophic, the Mission Operations Directorate initiated a measurement program in May 1990 to "better understand and manage these risks." The article discusses the reasons for program implementation, how metrics have aided in management decisions, which tools have been useful, and how the metrics have been used effectively.

To probe further

B. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, N.J., 1981. One of the first books to apply measurement to software engineering, it describes in detail a model for estimating effort and schedule from measurable project, process, and resource characteristics.

B. Curtis, "Measurement and experimentation in software engineering," *Proceedings of the IEEE*, Vol. 68, No. 9, 1980, pp. 1,144–1,157. This seminal paper includes the first description of the scales of measurement made in a software engineering context.

N. Fenton and S. Lawrence Pfleeger, *Software Metrics: A Rigorous Approach*, second edition, International Thomson Press, London, 1996. A thorough overview of software metrics, including measurement theory and descriptions of many product, process, and resource metrics. Includes new information about implementing a metrics program and validating software measures.

R. Grady and D. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, Englewood Cliffs, N.J., 1987. A clear and thorough description of the goals and activities of the Hewlett-Packard corporate measurement program.

R. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, Englewood Cliffs, N.J., 1992. A well-written guide to implementing a corporate metrics program, based on Grady's experience at Hewlett-Packard.

T. Khoshgoftaar and P. Oman (eds.), "Metrics in Software," *Computer*, Vol. 27, No. 9, Sept. 1994. A theme issue devoted to industrial applications of software metrics.

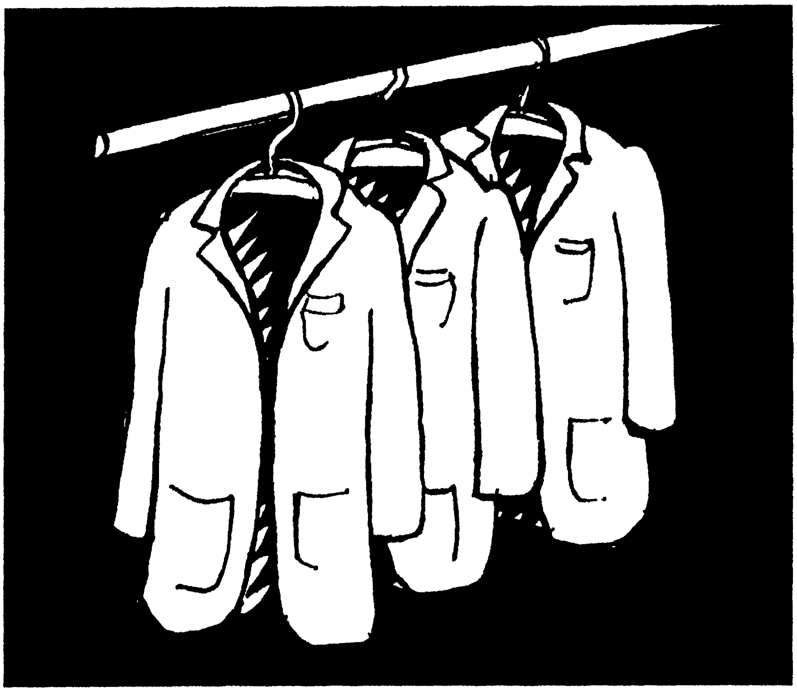
S. Lawrence Pfleeger, "Experimental Design and Analysis in Software Engineering," *Annals of Software Engineering*, Vol. 1, No. 1, 1995, pp. 219–253. The first of the published papers from the UK's DESMET project, it describes the basic issues in designing and carrying out software engineering experiments.

S. Lawrence Pfleeger and H. Dieter Rombach (eds.), "Measurement-Based Process Improvement," *IEEE Software*, Vol. 11, No. 4, July 1994. A theme issue devoted to software process improvement driven by measurement and metrics.

FEATURE

SCIENCE AND SUBSTANCE: A CHALLENGE TO SOFTWARE ENGINEERS

For 25 years, software researchers have proposed improving software development and maintenance with new practices whose effectiveness is rarely, if ever, backed up by hard evidence. We suggest several ways to address the problem, and we challenge the community to invest in being more scientific.



NORMAN FENTON and
SHARI LAWRENCE PFLEEGER
City University, London
ROBERT L. GLASS
Computing Trends

Software researchers and engineers are always seeking ways to improve their ability to build software. This search has resulted in such methods as

- structured design and programming,
- abstract data types,
- object-oriented design and program ming,
- CASE tools,
- statistical process control,
- maturity models,
- fourth-generation languages, and
- formal methods,

among others. But in spite of such "advances," software engineering in practice continues to be a labor-intensive, intellectually complex, and costly activity in which good management and communication seem to count for much more than technology.

At the same time, the January 1993 issue of the *IEEE CS Technical Committee on Software Engineering Newsletter* reported that since 1976 the Software Engineering Standards Committee of the IEEE Computer Society has developed 19 standards in the areas of terminology, requirements documentation, design documentation, user documentation, testing, verification and validation, reviews, and audits. And if you include all the major national standards bodies, there are in fact more than 250 software-engineering standards.

The existence of these standards raises some important questions. How do we know which practices to standardize? And are the standards not

Reprinted from *IEEE Software*, Vol. 11, No. 4, July 1994, pp. 86–95.

working or being ignored, since many development projects generate less-than-desirable products? The answer is that much of what we believe about which approaches are best is based on anecdotes, gut feelings, expert opinions, and flawed research, not on careful, rigorous software-engineering experimentation.

In this article, we examine some of the past and current problems with software-engineering research and technology transfer and suggest several ways to redirect our efforts toward improving our ability to build and maintain software.

RESEARCH CLAIMS

Developers who want to improve their productivity or the quality of their product are faced with an enormous choice of methods, tools, and standards. Adopting one or more often involves considerable time, expense, and trouble. Rational managers and their subordinates are prepared to invest in a new technology if they have evidence that using it will ultimately produce benefits. Although a single evaluation can never cover all possible situations, it is reasonable to seek some evidence of a new technology's likely *efficacy* when used under certain conditions.

But evidence is rare. Vendors' quantitative descriptions are often no more than sweeping claims like

- productivity gains of 250 percent,
- maintenance effort reduced by 80 percent, and
- integration time cut by five sixths.

Similar claims are often made by eminent experts. How can practitioners distinguish valid claims from invalid? And how can they determine that a particular method or technology is suited to their situation?

One way is to examine claims carefully from the viewpoint of scientific experimentation. As described by Vic Basili, Rick Selby, and David Hutchens in their classic paper on software-engineering experimentation, there *is* a scientifically sound way to design and carry out software-engineering investigations.[1] Their paper gives many examples of good research practice, plus guidelines for future experiments, but very few experiments reported since its publication have followed those recommendations.

Admittedly, experimentation in software engineering is notoriously difficult: Not only is it potentially expensive, but it can be daunting to try to control variables and environments. We applaud those who have performed an empirical study to confirm or refute their understanding of likely effects, even as we criticize certain experiments. Our intent is to suggest improvements to software-engineering research practices, in the hope that the results of future research will reflect a more solid scientific foundation. To do that, we compare good experiments with flawed ones, to illustrate the scrutiny required to determine if a recommended practice lives up to its claims.

HOW CAN YOU TELL IF CLAIMS ARE VALID? ASK FIVE QUESTIONS THAT ADDRESS EXPERIMENTAL TECHNIQUE.

RESEARCH REALITIES

Five questions should be (but rarely are) asked about any claim arising from software-engineering research:

- Is it based on empirical evaluation and data?
- Was the experiment designed correctly?
- Is it based on a toy or a real situation?
- Were the measurements used appropriate to the goals of the experiment?
- Was the experiment run for a long enough time?

Empiricism versus intuition. In many ways, software-engineering research got off to a bad start. Early researchers often assumed that if sufficient brilliance and analysis were put into conceiving a technique, benefits would surely follow. As a result, many research findings published can be characterized as "analytical advocacy research." That is, the authors describe a new concept in considerable detail, derive its potential benefits analytically, and recommend the concept be transferred to practice. Time passes, and other researchers derive similar conclusions from similar analyses. Eventually the consensus among researchers is that the concept has clear benefits. Yet practitioners often seem unenthused. Researchers, satisfied that their communal analysis is correct, become frustrated. Heated discussion and finger-pointing ensues.

Something important is missing from this picture: rigorous, quantitative experimentation. In the traditional scientific method used by researchers in other disciplines, the formulation of an idea and its related hypothesis is followed by evaluative research to investigate if the hypothesis is true or false. Only when research results confirm the hypothesis do researchers advocate broad-based technology transfer. Moreover, the research tries to quantify the magnitude, as well as the existence, of a benefit.

Evaluative research must involve realistic projects with realistic subjects, and it must be done with sufficient rigor to ensure that any benefits identified are clearly derived from the concept in question. This type of research is time-consuming and expensive and, admittedly, difficult to employ in all software-engineering research. It is not surprising that little of it is being done.

On the other hand, claims made by analytical advocacy are insupportable. Today, practitioners must place their

MEASUREMENT SCALES AND MEANINGFUL ANALYSIS

Measurement is the process of assigning a number or descriptor (a measure) to an entity to characterize a specific attribute of the entity. By manipulating these numbers, instead of the entities themselves, you make judgments about the entities. However, you must use the measures in mathematically correct ways if your judgments are to make sense. The type of measurement determines what analysis is acceptable.

Measurement types. You must assign measures that preserve your empirical observations about the attribute you are interested in. For example, if the attribute of the entity person that you want to measure is height, then you must assign a number to each person in a way that preserves empirical observations about height. If person A is taller than person B (an empirical observation), the measurement $M(A)$ must be greater than $M(B)$.

Sometimes there are many ways to assign numbers that preserve all empirical observations. For example, $M(A)$ is greater than $M(B)$ regardless of whether M is inches, feet, centimeters, or furlongs. Furthermore, the relationship among entities is preserved when you convert the attribute data from one measure to another, such as from inches to centimeters. Such a conversion is called an *admissible transformation.*

So any two valid measures, M and M', of the same attribute are related in a very specific way. For example, if M and M' are measures of height, there is always some constant c, greater than 0, such that $M = cM'$. If M is inches and M' is centimeters, then c is 2.54.

The kind of admissible transformations determines the measurement scale type. Height, for example, is a ratio scale type because multiplication is an admissible transformation. In general, the more restrictive the admissible transformations, the more sophisticated the scale type and the analyses that can be done. Table A defines the most common scale types, in increasing order of sophistication.

Usually, an attribute's scale type is not known a priori. Instead, you start with a crude understanding of an attribute, devise a simple way to measure it, accumulate data, and see if the results reflect the empirical behavior of the attribute. Then you clarify and reevaluate the attribute: Are you measuring what you really want to measure? This analysis helps you refine definitions and introduce new empirical relations, improving the accuracy of the measurement and, usually, increasing the sophistication of the measurement scale.

A goal of software measurement is to define measures that are on the most sophisticated scale possible, given the constraints of the real world. However, we still have only very crude empirical relations — and hence crude measurement scales — for attributes like software quality and productivity. Consider the software-failure attribute "criticality." Today we usually measure this by identifying different kinds of failures and relating them with a single binary relation, "is more critical than." This kind of empirical relational system defines a (relatively unsophisticated) ordinal scale type.

Meaningful measures. This formal definition of scale type based on admissible transformations lets you determine rigorously what kind of statements about your measurement are meaningful. Formally, a statement involving measurement is meaningful if its truth or falsity remains unchanged under any admissible transformation of the measures involved.

If you say "Fred is twice as tall as Jane," your statement implies that the measures are at least on the ratio scale, because multiplication is an admissible transformation. No matter which measure of height you use, the

faith in the reputation of the advocates who, although sometimes correct in the past, may not always be correct in the future. Consider the initial engineering attempts to allow humans to fly. Experts carefully studied the flight of birds, then developed flexible wings that would mimic it as closely as possible. This sounded fine in theory but was disastrous in practice. It was not until a completely new paradigm, using rigid wings and Bernoulli's laws, was conceived and tested that flight became possible. Empirical testing and analysis were critical to the discovery of the new paradigm.

Unfortunately, software methods and techniques often find their way into standards even when there is no reported empirical, quantitative evidence of their benefit. This is true of even the most sophisticated methods, developed with mathematical care and precision. For example, although there is some limited empirical evidence that fault-tolerant design for high-integrity systems (such as those that are safety-critical) is effective, there appears to be little or no published empirical work that supports the claims made on behalf of formal methods.

The case of formal methods is an especially interesting and instructive example of a revolutionary technique that has gained widespread appeal without rigorous experimentation. Formal methods are based on the use of mathematically precise specification and design notations. In its purest form, formal development is based on refinement and proof of correctness at each stage in the life cycle. In general, adopting formal methods requires a revolutionary change in development practices. There is no simple migration path, because the effective use of formal methods requires a radical change right at the beginning of the traditional life-cycle, when customer requirements are captured and recorded. Thus, the stakes are particularly high.

Yet, when Susan Gerhart, Dan Craigen, and Ted Ralston performed an extensive survey of formal methods use in industrial environments,[2] they concluded

> *There is no simple answer to the question: do formal methods pay off? Our cases provide a wealth of data but only scratch the surface of information available to address these questions. All cases involve so many interwoven factors that it is impossi-*

truth or falsity of the statement remains consistent.

But if you say, "The temperature in Tokyo today is twice that in London," your statement also implies the ratio scale, but in this case the ratio scale is not meaningful because air temperature is measured in Celsius and Fahrenheit. So, while it might be 40°C in Tokyo and 20°C in London (making your statement true), it would also be 104°F in Tokyo and 68°F in London (truth is not preserved). Thus, scalar multiplication is an inadmissible transformation, and this is an inappropriate use of measurement.

But suppose you said, "The difference in temperature between Tokyo and London today is twice what it was yesterday." This statement implies that the distance between two measures is meaningful, a condition that is part of the interval scale. The statement is meaningful, because Fahrenheit and Celsius are related by the affine transformation F = 9/5C + 32, which ensures that ratios of differences (as opposed to just ratios) are preserved. If it was 35°C yesterday in Tokyo and 25°C in London (a difference of 10) and today it is 40°C in Tokyo and 20°C in London (a difference of 20), the difference will be preserved when you transform the temperatures to the Fahrenheit scale: 95°F in Tokyo and 77°F London (a difference of 18) and 104°F in Tokyo and 68°F in London (a difference of 36).

Unfortunately, there are no such transformations for the software-failure attribute. The statement, "Failure x is twice as critical as failure y" is not meaningful because we have only an ordinal scale for failure criticality.

It is important to remember that meaningfulness is not the same as truth. Although the statement "Mickey Mouse is 102 years old" is clearly false, it is nevertheless a meaningful statement involving the age measure.

The notion of meaningfulness lets us determine what kind of operations we can perform on different measures. For example, it is meaningful to use the mean to compute the average of a data set measured on a ratio scale but not on an ordinal scale. Medians are meaningful for an ordinal scale but not for a nominal scale. These basic observations have been ignored in many software-measurement studies, in which a common mistake is to use the mean (rather than median) as the measure of average for data that is only ordinal.

TABLE A
COMMON SCALE TYPES

Scale type	Admissible transformations	Examples
Nominal	$M'=F(M)$ where F is any one-to-one mapping	Classification, for example software fault types (data, control, other)
Ordinal	$M'=F(M)$ where F is any monotonic increasing mapping that is, $M(x)\geq M(y)$ implies $M'(x)> M'(y)$	Ordering, for example, software failure by severity (negligible, marginal, critical, catastrophic)
Intervals	$M'=aM+b$ $(a>0)$	Calendar time, temperature (restricted to Fahrenheit and Celsius)
Ratio	$M'=aM$ $(a>0)$	Time interval, length
Absolute	$M'=M$	Counting

ble to allocate payoff from formal methods versus other factors, such as quality of people or effects of other methodologies. Even where data was collected, it was difficult to interpret the results across the background of the organization and the various factors surrounding the application.

One of the situations investigated by the Gerhart team was a joint project between IBM Hursley and the Programming Research Group at Oxford University.[3] For 12 years, this project used the Z specification language to respecify parts of Customer Information Control System-ESA Version 3 Release 1 as it was updated. The project made a serious attempt to quantify the benefits of using Z. As a result, the CICS project is widely believed to provide the best quantitative evidence to support the efficacy of formal methods, an observation confirmed by the Gerhart study.

The project would appear to be a huge success — so successful that IBM and PRG shared the prestigious Queen's Award for Technology. The project participants estimated that using Z reduced their costs by almost $5.5 million, a savings of nine percent overall. In addition, they claimed a 60 percent decrease in product failure rate. These results led the PGR's Geraint Jones to assert in his 1992 e-mail broadcast announcing the Queen's Award, "The moral of this tale is that formal methods cannot only improve quality, but also the timeliness and cost of producing state-of-the-art products." However, the quantified evidence to support these widely publicized claims is missing from the published results.

Another study casts doubt on the claim that formal methods are a universal solution to poor software quality. In a recent article, Peter Naur[4] reports that the use of formal notations does not lead inevitably to higher quality specifications, even when used by the most mathematically sophisticated minds. In his experiment, the use of a formal notation often led to more, not fewer, defects.

These studies suggest that the benefits of formal methods are not self-evident and argue for experiments. Yet there seems to be a widespread consensus that formal methods should be used on projects in which the software is safety-critical. For example, John McDermid[5] asserts that "these mathematical approaches provide us with the best available approach to the development of high-integrity safety-critical systems." In addition, the interim UK defense standard for such systems, DefStd 00-55, makes the use of formal methods mandatory.[6]

The assumption seems to be that no expense should be spared to improve confidence in the reliability of critical systems. Unfortunately, no real project has unlimited funds. Even safety-critical projects must use the most cost-effective way to ensure reliability. Rather than abandon formal methods, we suggest their use be embedded in the context of an experiment so that their effect on software quality and reliability can be studied and assessed. At present, there is no hard evidence to show that

- formal methods have been used cost-effectively on a realistic, safety-critical development;
- using formal methods delivers reliability more cost-effectively than, say, traditional structured methods with enhanced testing; and
- developers and users can be trained in sufficient numbers to use formal methods properly.

CURRICULA FOR THE MOST PART DO NOT COVER HOW TO ESTABLISH AND EVALUATE THE DESIGN OF EXPERIMENTS.

There is also the problem of choosing among competing formal methods, which we assume are not equally effective in a given situation. By thinking about a more scientific context before using formal methods, a project can try them and contribute to the larger body of software-engineering understanding.

There *are* some techniques that have become standards or standard practice after careful, empirical analysis. A good example is the use of inspections to uncover defects in code. Table 1 compares the efficiency of different kinds of testing techniques, as reported by Bob Grady.[7] This and similar research experiments confirm one of the few consensus views to emerge in empirical studies: Inspections are the cheapest and most effective testing techniques for finding faults.

Even here, it is important to keep the objective of the experiment in mind. The table shows overall testing efficiency, but does not report efficiency with respect to particular kinds of faults. Nevertheless, analyzing empirical data in the context of a rigorous investigation provides a sounder basis for changing practice than anecdote or intuition.

Experimental design. The experimental design must be correct for the hypothesis being tested. Some of the best publicized studies have subsequently been challenged on the basis of inappropriate experimental design. For example, an experiment by Ben Shneiderman and his colleagues showed that flowcharts did not help programmers comprehend documentation any better than pseudocode.[8] As a result, flowcharts were shunned in the software-engineering community and textbooks almost invariably use pseudocode instead of flowcharts to describe specific algorithms.

However, some years later David Scanlan demonstrated that structured flowcharts are preferable to pseudocode for program documentation.[9] Scanlan compared flowcharts and pseudocode with respect to the relative time needed to understand the algorithm and the relative time needed to make (accurate) changes to the algorithm. In both dimensions, flowcharts were clearly superior to pseudocode. Although some of Scanlan's criticisms of Shneiderman's study are controversial, he appears to have exposed a number of experimental flaws that explain the radically different conclusions about the two types of documentation. In particular, Scanlan demonstrated that Shneiderman overlooked several key variables in his experimental design.

Similar flaws in experimental design have misled the community about the benefits of structured programming. Harlan Mills' claims are typical:[10]

> *When a program was claimed to be 90 percent done with solid top-down structured programming, it would take only 10 percent more effort to complete it (instead of possibly another 90 percent!).*

But Iris Vessey and Ron Weber examined in detail the published empirical evidence to support the use of structured programming. They concluded that the evidence was "equivocal" and argued that the problems surrounding experimentation on structured programming are "a manifestation of poor theory, poor hypothesis, and poor methodology."[11]

The classic experiment by Gerald Weinberg on meeting goals shows that if you don't choose the attributes for determining success carefully, it is easy to maximize any single one as a success criterion.[12] Weinberg and Schulman gave each of six teams a different programming goal, and each team optimized its performance (and "succeeded") with respect to its goal — but performed poorly in terms of the other five goals. You can expect similar results if you run experiments out of context, because you will be narrowly defining "success" according to only one attribute.

These examples show that it is critical to examine experimental design carefully. Many software engineers are not familiar with how to establish or evaluate a proper design. This is due in no small part to the almost total absence of topics like experimental design, statistical analysis, and measurement principles in most computer-science and software-engineering curricula. The guidelines presented by Basili and his colleagues are a good first step, but the paper does not present important material in enough detail.

TABLE 1
COMPARISON OF TESTING EFFICIENCY

Testing type	Efficiency (defects found per hour)
Regular use	0.210
[illegible]	[illegible]
White box.322	
[illegible]	[illegible]

To address this problem, the British Department of Trade and Industry is now funding two projects in the UK: SMARTIE is producing guidelines about how to evaluate the effectiveness of standards and methods, and DESMET is preparing handbooks for software researchers and engineers on experimental design and statistical analysis.[13]

Toy versus real. Because of the cost of designing and running large-scale studies, exploratory research in software engineering is all too often conducted on artificial problems in artificial situations. Practitioners refer to these as toy projects in toy situations. The number of research studies using experienced practitioners (instead of students or novice programmers) on realistic projects is minuscule.

This is particularly noticeable in studies of programmers, a field in which evaluative and experimental research is the norm. At its major conference, Empirical Studies of Programmers, the community's leaders continue to recommend that researchers study real projects and real programmers, yet many of the findings reported at the conference continue to involve small, student projects. Because of cost and time constraints, even this community refrains from doing large-scale, realistic studies.

To be sure, evaluative research in the small is better than no evaluative research at all. And a small project may be appropriate for an initial foray into testing an idea or even a research design. For example, Vessey conducted an interesting experiment using students and small projects that indicates object orientation is not the natural approach to systems analysis and design that its advocates claim it to be.[14] The results are not conclusive, especially for experienced practitioners on real software projects, but it does indicate directions for further investigation. Similarly, Naur's experiment[4] was small but exposed a weakness in a popularly held belief about formal notations.

In another small but valuable study, Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich examined which looping constructs novice programmers found most natural.[15] Popular assumptions about structured programming are reflected in the fact that many languages supply a while-do loop (exit at the top) and a repeat-until loop (exit at the bottom). But the Soloway study revealed that the most natural looping structure was neither of these, but a loop that allows an exit in the middle, a technique disallowed in structured programming. This result implies that language designers, who followed common wisdom in not supplying such a loop, may inavertently make programming tasks more difficult than they need to be.

EXPERIMENTS MAY BE DESIGNED PROPERLY BUT MEASURE OR ANALYZE THE WRONG DATA.

How do the results from toy studies scale up to larger, more realistic situations? Although some studies have addressed this question (as we describe later in discussing Cleanroom), little research has been done to answer that question. The best that can be said is that, just as software-development-in-the-small differs from software-development-in-the-large, research-in-the-small may differ from research-in-the-large. There is something about the nature of software tasks and the required communication among team members that prevents our understanding of small-scale work from yielding an understanding of large-scale work.

Obviously, there is no easy solution to this problem. It is not possible for a lone researcher, operating on a relatively small budget, to conduct the kind of research needed. Credible studies require the cooperation and financial backing of major research institutions and software-development organizations. To date, such support has been rare.

Appropriate measures. Sometimes an experiment is designed properly but it measures and analyzes insufficient data or the wrong data.

Measuring the right attribute? The most common example is success criteria. For example, a study to demonstrate the effectiveness of using abstract data types used program size, measured in lines of code, as a measure of product quality.[16] Often purely subjective measures are used in the absence of objective measures. This is sometimes unavoidable; for example, in measuring user satisfaction. However, the conclusions you can draw from subjective data are very limited. For example, Virginia Gibson and James Senn[17] show that maintainers' subjective perceptions of which systems are most easily maintained differ wildly from objective data that measured maintainability.

Another measure that is commonly misleading is reliability. One of the most effective ways to demonstrate a method's efficacy is to show that it leads to more reliable software. How-

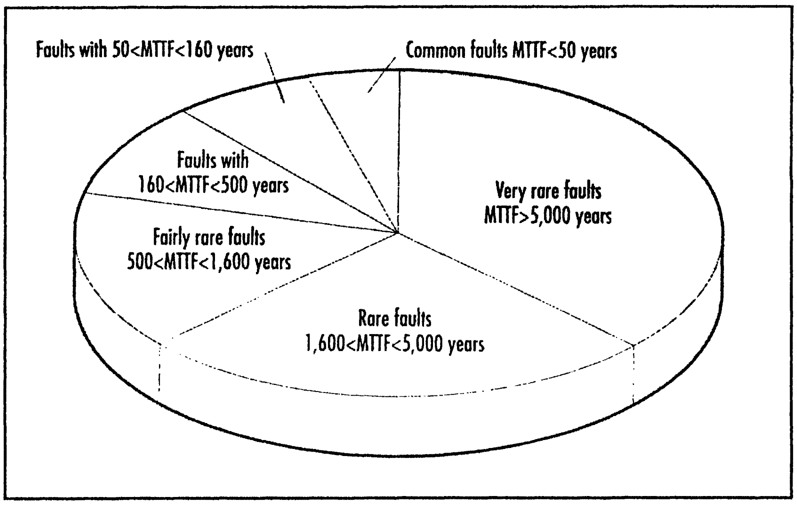


***Figure 1.** The relationship between faults and failures, which shows that focusing on faults instead of failures can be fatal. Studies that compare testing methods by using faults may be inappropriate and misleading.*

ever, measuring reliability involves tracking operational failures over time, and it is not always practical to wait until software is completed to evaluate its reliability. The most common "substitute" measure is the number of faults or defects discovered during development and testing, a number that can be very misleading.

At IBM, Ed Adams examined data from nine large software products, each with many thousands of years of logged use worldwide.[18] Figure 1 shows the relationship he discovered between detected faults and their manifestation as failures. For example, 33 percent of all faults led to a mean-time-to-failure greater than 5,000 years. In practical terms, such faults will almost never manifest as failures. Conversely, about two percent of faults led to an MTTF of less than 50 years. These faults are important to find, because a significant number of users will eventually be affected by the failures they cause.

It follows that finding and removing large numbers of faults may not necessarily improve reliability. The crucial task is to find the important two percent of faults. Thus, a focus on faults instead of failures can be fatal, unless a technique can identify the faults that have a short MTTF or greatly affect system behavior. Many studies have compared the effectiveness of different testing methods, but if the comparison is done in terms of general faults discovered, they may be inappropriate and misleading.

What scale? In addition to measuring the correct attribute, researchers must take care to evaluate and manipulate the measurements in a way that is appropriate to the design and the kind of data collected, as the box on pp. 88-89 briefly explains.

Data falls into one of five scales: nominal, ordinal, interval, ratio, and absolute. Each scale reflects the data's properties and can be manipulated only in certain ways. For example, nominal data includes labels or classifications, such as when you classify requirements as data requirements, interface requirements, and so on. Nominal data can be analyzed statistically in terms of frequency and mode, but not in terms of mean or median. In other words, only nonparametric statistical tests are valid on nominal data. The software-engineering literature is rife with experiments in which means and standard deviations are applied to nominal data, but their results are meaningless in the sense of formal measurement theory.

Likewise, there is an embarrassingly large set of literature in which inappropriate statistical techniques are applied. For example, a researcher might compare correlation coefficients across disparate sets of data instead of using the more appropriate analysis of variance. One of the most talked-about measures in software engineering is the Software Engineering Institute's process-maturity level. This five-point ordinal scale is only a valid measure of an organization's process maturity if it can be demonstrated that, in general, organizations at level $n + 1$ normally produce better software than organizations at level n. This relationship has not yet been demonstrated, although the SEI has told us that relevant studies are underway.

Long-term view. Sometimes research is designed and measured properly but just isn't carried on long enough. Short-term results masquerade as long-term effects. For example, speakers at the annual NASA Goddard Software Engineering Conference often report on an experiment at the Software Engineering Laboratory to investigate the benefits of using Ada instead of Fortran. The researchers examined a set of new Ada projects and found that the productivity and quality of the resulting Ada programs fell short of equivalent programs written in Fortran. However, the SEL did not stop there and report that Ada was a failure. It continued to develop programs in Ada, until each team had experience with at least three major Ada developments. These later results indicated that there were indeed significant benefits of Ada over Fortran.

The SEL concluded that the learning curve for Ada is long, and that the first set of projects represented programmers' efforts to code Fortran-like programs in Ada. By the third development, the programmers were taking advantage of Ada characteristics not available in Fortran, and these characteristics had measurable benefits. Thus, the long-term view led to conclusions very different from the short-term view.

The CASE Research Corp. found something similar when it considered the empirical evidence supporting the use of CASE tools.[19] They found that, contrary to the revolutionary improvements vendors invariably claimed, productivity normally decreased in the first year of CASE use, followed by modest improvement. Again, the short- and long-term assessments yielded opposite conclusions. However, the study found that the eventual improvement was rarely more than 10 percent and might be explained by factors other than the use of CASE (or may even fall within the margin of error). Moreover, compared with acquisition and upgrade costs, such modest improvements may indicate that CASE is not even cost-effective.

Researchers must take a long-term view of practices that promise to have a profound effect on development and maintenance, especially since the resistance of personnel to new techniques and the problems inherent in making radical changes quickly can mislead those who take only a short-term view.

RECENT EXAMPLES

Although most software-engineering research does not meet the requirements we outline here, some interesting examples do.

Cleanroom. Perhaps the single most complete research study involves Cleanroom.[20] Studies at the SEL, done in conjunction with the University of Maryland at College Park and Computer Sciences Corp., examined the Cleanroom error-detection and testing methodology using

- student subjects on small projects,
- NASA staff members on small real projects, and
- experienced industry practitioners on a sizable real project.

The findings used data collected both prestudy and within each context. For example, baseline data from projects not using the Cleanroom approach showed an error rate of six per thousand lines of code and productivity of 24 lines of code per day. The study of NASA staff using Cleanroom showed 4.5 errors per thousand LOC and productivity of 40 LOC per day, and the industry practitioners' Cleanroom project showed 3.2 errors per thousand LOC and productivity of 26 LOC per day. (Note how reliability improved significantly as Cleanroom was scaled up to a large program, but productivity did not.)

This study meets nearly all the criteria for good software-engineering research:

- It involved empirical evaluation and data.
- Its design was reasonable, given that the projects were "real."
- It involved both toy and real situations.
- The measurements were appropriate to the goals.
- The experiment was conducted over a period of time sufficient to encompass the effects of change in practice.

Object-oriented design. The SEL is also involved in a more mixed example of software-engineering research. In this case, it is gathering data over several years on eight major software projects using the object-oriented approach to building software. The series of studies is not finished, and the scaled-up study is not due for completion until 1996, but researchers are already reporting that the approaches studied represent "the most important methodology studies by the SEL to date."[21]

So far, researchers have reported that the amount of reuse rises dramatically when OO techniques are used, from 20 to 30 percent to 80 percent, and OO programs are about three-quarters the length (in lines of code) of comparable traditional solutions. On the other hand, OO projects have reported performance problems (although it is unclear how much of these problems are the result of OO), and OO appears to require significant domain analysis and project tailoring.

THERE ARE FAR TOO FEW EXAMPLES OF MODERATELY EFFECTIVE RESEARCH.

Unfortunately, the projects under study are also using Ada, and the studies have not separated the effects of OO from those of Ada. And because many of the benefits appear to be the result of increased reuse, it is not clear what gains are due to Ada, OO, or reuse.

So these studies meet many of, but not all, the goals for good research because

- They involve empirical evaluation and data.
- Use questionable experimental design.
- Involve real situations.
- Use measurements appropriate to the experimental goals.
- Are being run over an appropriate period of time.

4GLs. More typical of research approaches in the last decade are the studies of the benefits of fourth-generation languages. Several interesting studies published in the late 1980s compare Cobol and various 4GLs for implementing relatively simple business systems applications.[22-24] The findings of these studies are fascinating but hardly definitive. Some report productivity improving with the use of 4GLs by a factor of 4 to 5, while others describe only 29 to 39 percent differences. In some cases, object-code performance degraded by a factor of 15 to 174 for 4GLs, while other 4GLs produced code that was six times as fast!

It is apparent from the studies that measured effects are highly dependent on the 4GL studied, the project's application, and the people doing the job (for example, end users versus software specialists).

Examining the 4GL studies with the same criteria for good research in mind, we can make the following statements:

- The studies were based on empirical evidence and data.
- The experimental designs were reasonable.
- The projects were not toys, but neither were they sizable.
- The measurements were appropriate to the study goals.
- The experiments were not done over an extended period of time.

(Interestingly, two of the studies involved the same author, implying that the author may have made a second attempt at research in the topic area.)

Thus, recent examples of evaluative research paint a mixed picture. There are examples of effective research, but they are far too few in number. There are examples of moderately good research, and we can learn interesting things from them; however, follow-up, long-term, significant project studies are needed. And there are many examples of research that does no evaluation whatsoever. Given this spectrum, one thing is clear: there is considerable room for improvement.

We continue to look for new technologies to improve our ability to build and maintain software. But there is very little empirical evidence to confirm that technological fixes, such as introducing specific methods, tools and techniques, can radically improve the way we develop software systems. Even when improvements can be made by using specific methods, there is an urgent need to quantify the benefits and costs involved, and to compare these with competing technologies. At present, little quantitative data is available to help software managers make informed decisions about which method to use when change is needed.

The difficulty in performing the well-designed, quantitative assessments necessary to evaluate technologies in an objective manner is small compared with the massive resistance to change. Until there is widespread demand and expectation for objective measurement-based evaluation, software managers and standards bodies will continue to place their trust in unsubstantiated advertising claims, misleading or incomplete research reports, and anecdotal evidence.

Thus, we challenge the software-engineering community to take three major steps toward producing more rigorous and meaningful analyses of current and proposed practices:

♦ *For the software manager*: Insist on quantitative data and well-designed experimental research to substantiate any claims made for new or changed practices. And be willing to participate in such experiments to further your knowledge in particular and the software-engineering community's in general.

♦ *For the software developer or maintainer*: Be flexible and willing to participate in experiments involving existing or new techniques or methods. Try to be objective in providing data to researchers, and help them identify behaviors, attitudes, or practices that

ACKNOWLEDGMENTS

Norman Fenton is supported in part by the SMARTIE and PDCS2 projects. We thank Chris Kemerer, Bev Littlewood, Peter Mellor, and Stella Page for their contributions to this article. The final version was considerably improved as a result of the comments of several anonymous referees.

REFERENCES

1. V.R. Basili, R.W. Selby, and D.H. Hutchens, "Experimentation in Software Engineering," *IEEE Trans. Software Eng.*, June 1986, pp. 758-773.
2. S. Gerhart, D. Craigen, and A. Ralston, "Observation on Industrial Practice Using Formal Methods," *Proc. Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 24-33.
3. I. Houston and S. King, "CICS Project Report: Experiences and Results from the use of Z," *Lecture Notes in Computer Science*, Vol. 551, 1991.
4. P. Naur, "Understanding Turing's Universal Machine Personal Style in Program Description," *Computer J.*, No. 4, 1993, pp. 351-371.
5. J.A. McDermid, "Safety-Critical Software: A Vignette," *IEE Software Eng. J.*, No. 1, 1993, pp. 2-3.
6. *Interim Defence Standard 00-55: The Procurement of Safety-Critical Software in Defence Equipment*, Ministry of Defence Directorate of Standardization, Glasgow, Scotland, 1991.
7. R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs, N.J., 1992.
8. B. Shneiderman et al., "Experimental Investigations of the Utility of Detailed Flowcharts in Programming," *Comm. ACM*, June 1977, pp. 373-381.
9. D.A. Scanlan, "Structured Flowcharts Outperform Pseudocode: An Experimental Comparison," *IEEE Software*, Sept. 1989, pp. 28-36.
10. H. Mills, "Structured Programming: Retrospect and Prospect," *IEEE Software*, Nov. 1986, pp. 58-66.
11. I. Vessey and R. Weber, "Research on Structured Programming: An Empiricist's Evaluation," *IEEE Trans. Software Eng.*, July 1984, pp. 397-407.
12. G. Weinberg and E. Schulman, "Goals and Performance in Computer Programming," *Human Factors*, No. 1, 1974, pp. 70-77.
13. W.-E. Mohamed, C.J. Sadler, and D. Law, "Experimentation in Software Engineering: A New Framework," *Proc. Software Quality Management '93*, Elsevier Science, Essex, U.K. and Computational Mechanics Publications, Southampton, U.K., 1993.
14. I. Vessey and S. Conger, "Requirements Specification: Learning Object, Process, and Data Methodologies," *Comm. ACM*, May 1994.
15. E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive Strategies and Looping Constructs: An Empirical Survey," *Comm. ACM*, Nov. 1983, pp. 853-860.
16. J. Mitchell, J.E. Urban, and R. McDonald, "The Effect of Abstract Data Types on Program Development," *Computer*, Aug. 1987, pp. 85-88.
17. V.R. Gibson and J.A. Senn, "System Structure and Software Maintenance Performance," *Comm. ACM*, Mar. 1989, pp. 347-358.
18. E. Adams, "Optimizing Preventive Service of Software Products," *IBM J. Research and Development*, No. 1, 1984, pp. 2-14.
19. CASE Research Group, *The Second Annual Report on CASE*, Bellevue, Wash., 1990.
20. A. Kouchakdjian and V.R. Basili, "Evaluation of the Cleanroom Methodology in the SEL," *Proc. Software Eng. Workshop*, NASA Goddard, Greenbelt, MD., 1989.
21. M. Stark, "Impacts of Object-Oriented Technologies: Seven Years of Software Engineering," *J. Systems and Software*, Nov. 1993.
22. S.K. Misra and P.J. Jalics, "Third-Generation versus Fourth-Generation Software Development," *IEEE Software*, July 1988, pp. 8-14.
23. V. Matos and P.J. Jalics, "An Experimental Analysis of the Performance of Fourth-Generation Tools on PCs," *Comm. ACM*, Nov. 1989.
24. J. Verner and G. Tate, "Estimating Size and Effort in Fourth-Generation Development," *IEEE Software*, July 1988, pp. 15-22.

might affect the aspects of the project being studied.

◆ *For the software researcher*: Employ evaluative research as a necessary component in exploring new ideas. Learn about rigorous experimentation, and design your projects accordingly. Try to quantify as much as possible, and identify the degree to which you have control over each of the variables you are studying.

By taking these steps, the entire community should benefit. Finding willing industrial partners for research should be made easier, as the potential benefit to all participants is clear. The European Community has recognized the urgent need for quantitative evaluation performed by industry-research partnerships. A new program called the European Systems and Software Initiative has been defined and funded (initial funding is $50 million) to support projects that aim to evaluate specific software methods or tools. Eventually, with programs such as these, the practice of software engineering will benefit from better approaches resulting from scientific investigation and demonstrated improvement. ◆

Experimentation in Software Engineering

VICTOR R. BASILI, SENIOR MEMBER, IEEE, RICHARD W. SELBY, MEMBER, IEEE, AND DAVID H. HUTCHENS, MEMBER, IEEE

Abstract—Experimentation in software engineering supports the advancement of the field through an iterative learning process. In this paper we present a framework for analyzing most of the experimental work performed in software engineering over the past several years. We describe a variety of experiments in the framework and discuss their contribution to the software engineering discipline. Some useful recommendations for the application of the experimental process in software engineering are included.

Index Terms—Controlled experiment, data collection and analysis, empirical study, experimental design, software metrics, software technology measurement and evaluation.

I. Introduction

As any area matures, there is the need to understand its components and their relationships. An experimental process provides a basis for the needed advancement in knowledge and understanding. Since software engineering is in its adolescence, it is certainly a candidate for the experimental method of analysis. Experimentation is performed in order to help us better evaluate, predict, understand, control, and improve the software development process and product.

Experimentation in software engineering, as with any other experimental procedure, involves an iteration of a hypothesize and test process. Models of the software process or product are built, hypotheses about these models are tested, and the information learned is used to refine the old hypotheses or develop new ones. In an area like software engineering, this approach takes on special importance because we greatly need to improve our knowledge of how software is developed, the effect of various technologies, and what areas most need improvement. There is a great deal to be learned and intuition is not always the best teacher.

In this paper we lay out a framework for analyzing most of the experimental work that has been performed in software engineering over the past several years. We then discuss a variety of these experiments, their results, and the impact they have had on our knowledge of the software engineering discipline.

II. Objectives

There are three overall goals for this work. The first objective is to describe a framework for experimentation in software engineering. The framework for experimentation is intended to help structure the experimental process and to provide a classification scheme for understanding and evaluating experimental studies. The second objective is to classify and discuss a variety of experiments from the literature according to the framework. The description of several software engineering studies is intended to provide an overview of the knowledge resulting from experimental work, a summary of current research directions, and a basis for learning from past experience with experimentation. The third objective is to identify problem areas and lessons learned in experimentation in software engineering. The presentation of problem areas and lessons learned is intended to focus attention on general trends in the field and to provide the experimenter with useful recommendations for performing future studies. The following three sections address these goals.

III. Experimentation Framework

The framework of experimentation, summarized in Fig. 1, consists of four categories corresponding to phases of the experimentation process: 1) definition, 2) planning, 3) operation, and 4) interpretation. The following sections discuss each of these four phases.

A. Experiment Definition

The first phase of the experimental process is the study definition phase. The study definition phase contains six parts: 1) motivation, 2) object, 3) purpose, 4) perspective, 5) domain, and 6) scope. Most study definitions contain each of the six parts; an example definition appears in Fig. 2.

There can be several motivations, objects, purposes, or perspectives in an experimental study. For example, the motivation of a study may be to understand, assess, or improve the effect of a certain technology. The "object of study" is the primary entity examined in a study. A study may examine the final software product, a development process (e.g., inspection process, change process), a model (e.g., software reliability model), etc. The

Manuscript received July 15, 1985; revised January 15, 1986. This work was supported in part by the Air Force Office of Scientific Research under Contract AFOSR-F49620-80-C-001 and by the National Aeronautics and Space Administration under Grant NSG-5123 to the University of Maryland. Computer support was provided in part by the Computer Science Center at the University of Maryland.

V. R. Basili is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

R. W. Selby was with the Department of Computer Science, University of Maryland, College Park, MD 20742. He is now with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

D. H. Hutchens is with the Department of Computer Science, Clemson University, Clemson, SC 29634.

IEEE Log Number 8608188.

Reprinted from *IEEE Trans. Software Eng.*, Vol. SE-12, No. 7, July 1986, pp. 733–743.

I. Definition					
Motivation	Object	Purpose	Perspective	Domain	Scope
Understand Assess Manage Engineer Learn Improve Validate Assure	Product Process Model Metric Theory	Characterise Evaluate Predict Motivate	Developer Modifier Maintainer Project manager Corporate manager Customer User Researcher	Programmer Program/project	Single project Multi-project Replicated project Blocked subject-project

II. Planning		
Design	Criteria	Measurement
Experimental designs Incomplete block Completely randomised Randomized block Fractional factorial Multivariate analysis Correlation Factor analysis Regression Statistical models Non-parametric Sampling	Direct reflections of cost/quality Cost Errors Changes Reliability Correctness Indirect reflections of cost/quality Data coupling Information visibility Programmer comprehension Execution coverage Sise Complexity	Metric definition Goal-question-metric Factor-criteria-metric Metric validation Data collection Automatability Form design and test Objective vs. subjective Level of measurement Nominal/classificatory Ordinal/ranking Interval Ratio

III. Operation		
Preparation	Execution	Analysis
Pilot study	Data collection Data validation	Quantitative vs. qualitative Preliminary data analysis Plots and histograms Model assumptions Primary data analysis Model application

IV. Interpretation		
Interpretation context	Extrapolation	Impact
Statistical framework Study purpose Field of research	Sample representativeness	Visibility Replication Application

Fig. 1. Summary of the framework of experimentation.

Definition element	example
Motivation	To improve the unit testing process,
Purpose	characterize and evaluate
Object	the processes of functional and structural testing
Perspective	from the perspective of the developer
Domain: programmer	as they are applied by experienced programmers
Domain: program	to unit-sise software
Scope	in a blocked subject-project study.

Fig. 2. Study definition example.

#Teams per project	#Projects	
	one	more than one
one	Single project	Multi-project variation
more than one	Replicated project	Blocked subject-project

Fig. 3. Experimental scopes.

purpose of a study may be to characterize the change in a system over time, to evaluate the effectiveness of testing processes, to predict system development cost by using a cost model, to motivate[1] the validity of a theory by analyzing empirical evidence, etc. In experimental studies that examine "software quality," the interpretation usually includes correctness if it is from the perspective of a developer or reliability if it is from the perspective of a customer. Studies that examine metrics for a given project type from the perspective of the project manager may interest certain project managers, while corporate managers may only be interested if the metrics apply across several project types.

Two important domains that are considered in experimental studies of software are 1) the individual programmers or programming teams (the "teams") and 2) the programs or projects (the "projects"). "Teams" are (possibly single-person) groups that work separately, and "projects" are separate programs or problems on which teams work. Teams may be characterized by experience, size, organization, etc., and projects may be characterized by size, complexity, application, etc. A general classification of the scopes of experimental studies can be obtained by examining the sizes of these two domains considered (see Fig. 3). Blocked subject-project studies examine one or more objects across a set of teams and a set of projects. Replicated project studies examine ob-

[1] For clarification, the usage of the word "motivate" as a study purpose is distinct from the study "motivation."

ject(s) across a set of teams and a single project, while multiproject variation studies examine object(s) across a single team and a set of projects. Single project studies examine object(s) on a single team and a single project. As the representativeness of the samples examined and the scope of examination increase, the wider-reaching a study's conclusions become.

B. Experiment Planning

The second phase of the experimental process is the study planning phase. The following sections discuss aspects of the experiment planning phase: 1) design, 2) criteria, and 3) measurement.

The design of an experiment couples the study scope with analytical methods and indicates the domain samples to be examined. Fractional factorial or randomized block designs usually apply in blocked subject-project studies, while completely randomized or incomplete block designs usually apply in multiproject and replicated project studies [33], [41]. Multivariate analysis methods, including correlation, factor analysis, and regression [75], [80], [89], generally may be used across all experimental scopes. Statistical models may be formulated and customized as appropriate [89]. Nonparametric methods should be planned when only limited data may be available or distributional assumptions may not be met [100]. Sampling techniques [40] may be used to select representative programmers and programs/projects to examine.

Different motivations, objects, purposes, perspectives, domains, and scopes require the examination of different criteria. Criteria that tend to be direct reflections of cost/quality include cost [114], [108], [86], [5], [28], errors/changes [49], [24], [112], [2], [81], [13], reliability [42], [64], [56], [69], [70], [76], [77], [95], and correctness [51], [61], [68]. Criteria that tend to be indirect reflections of cost/quality include data coupling [62], [48], [104], [78], information visibility [85], [83], [55], programmer understanding [99], [103], [109], [113], execution coverage [105], [15], [18], and size/complexity [11], [59], [71].

The concrete manifestations of the cost/quality aspects examined in the experiment are captured through measurement. Paradigms assist in the metric definition process: the goal-question-metric paradigm [17], [25], [19], [93] and the factor-criteria-metric paradigm [39], [72]. Once appropriate metrics have been defined, they may be validated to show that they capture what is intended [7], [21], [45], [50], [108], [116]. The data collection process includes developing automated collection schemes [16] and designing and testing data collection forms [25], [27]. The required data may include both objective and subjective data and different levels of measurement: nominal (or classificatory), ordinal (or ranking), interval, or ratio [100].

C. Experiment Operation

The third phase of the experimental process is the study operation phase. The operation of the experiment consists of 1) preparation, 2) execution, and 3) analysis. Before conducting the actual experiment, preparation may include a pilot study to confirm the experimental scenario, help organize experimental factors (e.g., subject expertise), or inoculate the subjects [45], [44], [63], [18], [113], [73]. Experimenters collect and validate the defined data during the execution of the study [21], [112]. The analysis of the data may include a combination of quantitative and qualitative methods [30]. The preliminary screening of the data, probably using plots and histograms, usually precedes the formal data analysis. The process of analyzing the data requires the investigation of any underlying assumptions (e.g., distributional) before the application of the statistical models and tests.

D. Experiment Interpretation

The fourth phase of the experimental process is the study interpretation phase. The interpretation of the experiment consists of 1) interpretation context, 2) extrapolation, and 3) impact. The results of the data analysis from a study are interpreted in a broadening series of contexts. These contexts of interpretation are the statistical framework in which the result is derived, the purpose of the particular study, and the knowledge in the field of research [16]. The representativeness of the sampling analyzed in a study qualifies the extrapolation of the results to other environments [17]. Several follow-up activities contribute to the impact of a study: presenting/publishing the results for feedback, replicating the experiment [33], [41], and actually applying the results by modifying methods for software development, maintenance, management, and research.

IV. Classification of Analyses

Several investigators have published studies in the four general scopes of examination: blocked subject-project, replicated project, multiproject variation, or single project. The following sections cite studies from each of these categories. Note that surveys on experimentation methodology in empirical studies include [35], [96], [74], [98]. Each of the sections first discusses one experiment in moderate depth, using italicized keywords from the framework for experimentation, and then chronologically presents an overview of several others in the category. In any survey of this type it is almost certain that some deserving work has been accidentally omitted. For this, we apologize in advance.

A. Blocked Subject-Project Studies

With a *motivation* to improve and better understand unit testing, Basili and Selby [18] conducted a study whose *purpose* was to characterize and evaluate the processes (i.e., *objects*) of code reading, functional testing, and structural testing from the *perspective* of the developer. The testing processes were examined in a blocked subject-project *scope*, where 74 student through professional programmers (from the programmer *domain*) tested four unit-size programs (from the program *domain*) in a replicated fractional factorial *design*. Objective *measurement* of the testing processes was in several *criteria* areas: fault detection effectiveness, fault detection cost, and classes of faults detected. Experiment *preparation* included a pilot study [63], *execution* incorporated both manual and automated monitoring of testing activity, and *analysis* used analysis of variance methods [33], [90]. The major results (in the *interpretation context* of the study purpose) included: 1) with the professionals, code reading detected more software faults and had a higher fault detection rate than did the other methods; 2) with the professionals, functional testing detected more faults than did structural testing, but they were not different in fault detection rate; 3) with the students, the three techniques were not different in performance, except that structural testing detected fewer faults than did the others in one study phase; and 4) overall, code reading detected more interface faults and functional testing detected more control faults than did the other methods. A major result (in the *interpretation context* of the field of research) was that the study suggested that nonexecution based fault detection, as in code reading, is at least as effective as on-line methods. The particular programmers and programs sampled qualify the *extrapolation* of the results. The *impact* of the study was an advancement in the understanding of effective software testing methods.

In order to understand program debugging, Gould and Drongowski [58] evaluated several related factors, including effect of debugging aids, effect of fault type, and effect of particular program debugged from the perspective of the developer and maintainer. Thirty experienced programmers independently debugged one of four one-page programs that contained a single fault from one of three classes. The major results of these studies were: 1) debugging is much faster if the programmer has had previous experience with the program, 2) assignment bugs were harder to find than other kinds, and 3) debugging aids did not seem to help programmers debug faster. Consistent results were obtained when the study was conducted on ten additional experienced programmers [57]. These results and the identification of possible "principles" of debugging contributed to the understanding of debugging methodology.

In order to improve experimentation methodology and its application, Weissman [113] evaluated programmers' ability to understand and modify a program from the perspective of the developer and modifier. Various measures of programmer understanding were calculated, in a series of factorial design experiments, on groups of 16-48 university students performing tasks on two small programs. The study emphasized the need for well-structured and well-documented programs and provided valuable testimony on and worked toward a suitable experimentation methodology.

In order to assess the impact of language features on the programming process, Gannon and Horning [54] characterized the relationship of language features to software reliability from the perspective of the developer. Based on an analysis of the deficiencies in a programming language, nine different features were modified to produce a new version. Fifty-one advanced students were divided into two groups and asked to complete implementations of two small but sophisticated programs (75-200 line) in the original language and its modified version. The redesigned features in the two languages were contrasted in program fault frequency, type, and persistence. The experiment identified several language-design decisions that significantly affected reliability, which contributed to the understanding of language design for reliable software.

In order to understand the unit testing process better, Hetzel [60] evaluated a reading technique and functional and "selective" testing (a composite approach) from the perspective of the developer. Thirty-nine university students applied the techniques to three unit-size programs in a Latin square design. Functional and "selective" testing were equally effective and both superior to the reading technique, which contributed to our understanding of testing methodology.

In order to improve and better understand the maintenance process, Curtis *et al.* [44] conducted two experiments to evaluate factors that influence two aspects of software maintenance, program understanding, and modification, from the perspective of the developer and maintainer. Thirty-six junior through advanced professional programmers in each experiment examined three classes of small (36-57 source line) programs in a factorial design. The factors examined include control flow complexity, variable name mnemonicity, type of modification, degree of commenting, and the relationship of programmer performance to various complexity metrics. In [45] they continued the investigation of how software characteristics relate to psychological complexity and presented a third experiment to evaluate the ability of 54 professional programmers to detect program bugs in three programs in a factorial design. The series of experiments suggested that software science [59] and cyclomatic complexity [71] measures were related to the difficulty experienced by programmers in locating errors in code.

In order to improve and better understand program debugging, Weiser [110] evaluated the theory that "programmers use 'slicing' (stripping away a program's statements that do not influence a given variable at a given statement) when debugging" from the perspective of the developer, maintainer, and researcher. Twenty-one university graduate students and programming staff debugged a fault in three unit-size (75-150 source line) programs in a nonparametric design. The study results supported the slicing theory, that is, programmers during debugging routinely partitioned programs into a coherent, discontiguous piece (or slice). The results advanced the understanding of software debugging methodology.

In order to improve design techniques, Ramsey, Atwood, and Van Doren [87] evaluated flowcharts and program design languages (PDL) from the perspective of the developer. Twenty-two graduate students designed two small (approximately 1000 source line) projects, one using flowcharts and the other using PDL. Overall, the results suggested that design performance and designer-programmer communication were better for projects using PDL.

In order to validate a theory of programming knowledge, Soloway and Ehrlich [102] conducted two studies, using 139 novices and 41 professional programmers, to evaluate programmer behavior from the perspective of the researcher. The theory was that programming knowledge contained programming plans (generic program fragments representing common sequences of actions) and rules of programming discourse (conventions used in composing plans into programs). The results supported the existence and use of such plans and rules by both novice and advanced programmers.

Other blocked subject-project studies include [82], [115], and [111].

B. Replicated Project Studies

With a *motivation* to assess and better understand team software development methodologies, Basili and Reiter [16] conducted a study whose *purpose* was to characterize and evaluate the development processes (i.e., *objects*) of a 1) disciplined-methodology team approach, 2) ad hoc team approach, and 3) ad hoc individual approach from the *perspective* of the developer and project manager. The development processes were examined in a replicated project *scope*, in which advanced university students comprising seven three-person teams, six three-person teams, and six individuals (from the programmer *domain*) used the approaches, respectively. They separately developed a small (600–2200 line) compiler (from the program *domain*) in a nonparametric *design*. Objective *measurement* of the development approaches was in several *criteria* areas: number of changes, number of program runs, program data usage, program data coupling/binding, static program size/complexity metrics, language usage, and modularity. Experiment *preparation* included presentation of relevant material [68], [8], [34], *execution* included automated monitoring of on-line development activity and *analysis* used nonparametric comparison methods. The major results (in the *interpretation context* of the study purpose) included: 1) the methodological discipline was a key influence on the general efficiency of the software development process; 2) the disciplined team methodology significantly reduced the costs of software development as reflected in program runs and changes; and 3) the examination of the effect of the development approaches was accomplished by the use of quantitative, objective, unobtrusive, and automatable process and product metrics. A major result (in the *interpretation context* of the field of research) was that the study supported the belief that incorporating discipline in software development reflects positively on both the development process and final product. The particular programmers and program sampled qualify the *extrapolation* of the results. The *impact* of the study was an advancement in the understanding of software development methodologies and their evaluation.

In order to improve the design and implementation processes, Parnas [84] evaluated system modularity from the perspective of the developer. Twenty university undergraduates each developed one of four different types of implementations for one of five different small modules. Then each of the modules were combined with others to form several versions of the whole system. The results were that minor effort was required in assembling the systems and that major system changes were confined to small, well-defined subsystems. The results supported the ideas on formal specifications and modularity discussed in [83] and [85], and advanced the understanding of design methodology.

In order to assess the impact of static typing of programming languages in the development process, Gannon [53] evaluated the use of a statically typed language (having integers and strings) and a "typeless" language (e.g., arbitrary subscripting of memory) from the perspective of the developer. Thirty-eight students programmed a small (48–297 source line) problem in both languages, with half doing it in each order. The two languages were compared in the resulting program faults, the number of runs containing faults, and the relation of subject experience to fault proneness. The major result was that the use of a statically typed language can increase programming reliability, which improved our understanding of the design and use of programming languages.

In order to improve program composition, comprehension, debugging, and modification, Shneiderman [99] evaluated the use of detailed flowcharts in these tasks from the perspective of the developer, maintainer, modifier, and researcher. Groups of 53–70 novice through intermediate subjects, in a series of five experiments, performed various tasks using small programs. No significant differences were found between groups that used and those that did not use flowcharts, questioning the merit of using detailed flowcharts.

In order to improve and better understand the unit testing process, Myers [79] evaluated the techniques of three-person walk-throughs, functional testing, and a control group from the perspective of the developer. Fifty-nine junior through advanced professional programmers applied the techniques to test a small (100 source line) but nontrivial program. The techniques were not different in the number of faults they detected, all pairings of techniques were superior to single techniques, and code reviews were less cost-effective than the others. These results improved our understanding of the selection of appropriate software testing techniques.

In order to validate a particular metric family, Basili and Hutchens [11] evaluated the ability of a proposed metric family to explain differences in system development methodologies and system changes from the perspective of the developer, project manager, and researcher. The metrics were applied to 19 versions of a small (600–2200) compiler, which were developed by teams of advanced university students using three different development approaches (see the first study [16] described in this section). The major results included: 1) the metrics were able to differentiate among projects developed with different development methodologies; and 2) the differences among individuals had a large effect on the relationships between the metrics and aspects of system development. These results provided insights into the formulation and appropriate use of software metrics.

In order to improve the understanding of why software errors occur, Soloway *et al.* [65], [101] characterized programmer misconceptions, cognitive strategies, and their manifestations as bugs in programs from the perspective of the developer and researcher. Two hundred and four novice programmers separately attempted implementations of an elementary program. The results supported the programmers' intended use of "programming plans" [103] and revealed that most people preferred a read-process strategy over a process-read strategy. The results advanced the understanding of how individuals write programs, why they sometimes make errors, and what programming language constructs should be available.

In order to understand the effect of coding conventions on program comprehensibility, Miara *et al.* [73] conducted a study to evaluate the relationship between indentation levels and program comprehension from the perspective of the developer. Eighty-six novice through professional subjects answered questions about one of seven program variations with different level and type of indentation. The major result was that an indentation level of two or four spaces was preferred over zero or six spaces.

In order to improve software development approaches, Boehm, Gray, and Seewaldt [29] characterized and evaluated the prototyping and specifying development approaches from the perspective of the developer, project manager, and user. Seven two- and three-person teams, consisting of university graduate students, developed versions of the same application software system (2000-4000 line); four teams used a requirement/design specifying approach and three teams used a prototyping approach. The systems developed by prototyping were smaller, required less development effort, and were easier to use. The systems developed by specifying had more coherent designs, more complete functionality, and software that was easier to integrate. These results contributed to the understanding of the merits and appropriateness of software development approaches.

In order to validate the theoretical model for *N*-version programming [3], [66], Knight and Leveson [67] conducted a study to evaluate the effectiveness of *N*-version programming for reliability from the perspective of the customer and user. *N*-version programming uses a high-level driver to connect several separately designed versions of the same system, the systems "vote" on the correct solution, and the solution provided by the majority of the systems is output. Twenty-seven graduate students were asked to independently design an 800 source line system. The factors examined included individual system reliability, total *N*-version system reliability, and classes of faults that occurred in systems simultaneously. The major result was that the assumption of independence of the faults in the programs was not justified, and therefore, the reliability of the combined "voting" system was not as high as given by the model.

In order to improve and better understand software development approaches, Selby, Basili, and Baker [94] characterized and evaluated the Cleanroom development approach [46], [47], in which software is developed without execution (i.e., completely off-line), from the perspective of the developer, project manager, and customer. Fifteen three-person teams of advanced university students separately developed a small system (800-2300 source line); ten teams used Cleanroom and five teams used a traditional development approach in a nonparametric design. The major results included: 1) most developers using the Cleanroom approach were able to build systems without program execution; and 2) the Cleanroom teams' products met system requirements more completely and succeeded on more operational test cases than did those developed with a traditional approach. The results suggested the feasibility of complete off-line development, as in Cleanroom, and advanced the understanding of software development methodology.

Other replicated project studies include [37], [4], and [63].

C. Multiproject Variation Studies

With a *motivation* to improve the understanding of resource usage during software development, Bailey and Basili [5] conducted a study whose *purpose* was to predict development cost by using a particular model (i.e., *object*) and to evaluate it from the *perspective* of the project manager, corporate manager, and researcher. The particular model generation method was examined in a multiproject *scope*, with baseline data from 18 large (2500-100 000 source line) software projects in the NASA S.E.L. [27], [26], [38], [91] production environment (from the program *domain*), in which teams contained from two to ten programmers (from the programmer *domain*). The study *design* incorporated multivariate methods to parameterize the model. Objective and subjective *measurement* of the projects was based on 21 *criteria*[2] in three areas: methodology, complexity, and personnel experience. Study *preparation* included preliminary work [52], *execution* included an established set of data collection forms [27], and *analysis* used forward multivariate regression methods. The major results (in the *interpretation context* of the study purpose) included 1) the estimation of software development resource usage improved by considering a set of both baseline and customization factors; 2) the application in the NASA environment of

[2] Twenty-one factors were selected after examining a total of 82 factors that possibly contributed to project resource expenditure, including 36 from [108] and 16 from [28].

the proposed model generation method, which considers both types of factors, produced a resource usage estimate for a future project within one standard deviation of the actual; and 3) the confirmation of the NASA S.E.L. formula that the cost per line of reusing code is 20 percent of that of developing new code. A major result (in the *interpretation context* of the field of research) was that the study highlighted the difference of each software development environment, which improved the selection and use of resource estimation models. The particular programming environment and projects sampled qualify the *extrapolation* of the results. The *impact* of the study was an advancement in the understanding of estimating software development resource expenditure.

In order to assess, manage, and improve multiproject environments, several researchers [28], [20], [108], [10], [36], [21], [62], [112], [97], [107] have characterized, evaluated, and/or predicted the effect of several factors from the perspective of the developer, modifier, project manager, and corporate manager. All the studies examined moderate to large projects from production environments. The relationships investigated were among various factors, including structured programming, personnel background, development process and product constraints, project complexity, human and computer resource consumption, error-prone software identification, error/change distributions, data coupling/binding, project duration, staff size, degree of management control, and productivity. These studies have provided increased project visibility, greater understanding of classes of factors sensitive to project performance, awareness of the need for project measurement, and efforts for standardization of definitions. Analysis has begun on incorporating project variation information into a management tool [9], [14].

In order to improve and better understand the software maintenance process, Vessey and Weber [106] conducted an experiment to evaluate the relationship between the rate of maintenance repair and various product and process metrics from the perspective of the developer, user, and the project manager. A total of 447 small (up to 600 statements) commercial and clerical Cobol programs from one Australian organization and two U.S. organizations were analyzed. The product and process metrics included program complexity, programming style, programmer quality, and number of system releases. The major results were: 1) in the Australian organization, program complexity and programming style significantly affected the maintenance repair rate; and 2) in the U.S. organizations, the number of times a system was released significantly affected the maintenance repair rate.

In order to improve the software maintenance process, Adams [1] evaluated operational faults from the perspective of the user, customer, project manager, and corporate manager. The fault history for nine large production products (e.g., operating system releases or their major components) were empirically modeled. He developed an approach for estimating whether and under what circumstances preventively fixing faults in operational software in the field was appropriate. Preventiv:ly fixing faults consisted of installing fixes to faults that had yet to be discovered by particular users, but had been discovered by the vendor or other users. The major result was that for the typical user, corrective service was a reasonable way of dealing with most faults after the code had been in use for a fairly long period of time, while preventively fixing high-rate faults was advantageous during the time immediately following initial release.

In order to assess the effectiveness of the testing process, Bowen [31] evaluated estimations of the number of residual faults in a system from the perspective of the customer, developer, and project manager. The study was based on fault data collected from three large (2000–6000 module) systems developed in the Hughes–Fullerton environment. The study partitioned the faults based on severity and analyzed the differences in estimates of remaining faults according to stage of testing. Insights were gained into relationships between fault detection rates and residual faults.

D. Single Project Studies

With a *motivation* to improve software development methodology, Basili and Turner [22] conducted a study whose *purpose* was to characterize the process (i.e., *object*) of iterative enhancement in conjunction with a top-down, stepwise refinement development approach from the *perspective* of the developer. The development process was examined in a single project *scope*, where the authors, two experienced individuals (from the programmer *domain*), built a 17 000 line compiler (from the program *domain*). The study *design* incorporated descriptive methods to capture system evolution. Objective *measurement* of the system was in several *criteria* areas: size, modularity, local/global data usage, and data binding/coupling [62], [104]. Study *preparation* included language design [23], *execution* incorporated static analysis of system snapshots, and *analysis* used descriptive statistics. The results (in the *interpretation context* of the statistical framework) included: 1) the percentage of global variables decreased over time while the percentage of actual versus possible data couplings across modules increased, suggesting the usage of global data became more appropriate over time; and 2) the number of procedures and functions rose over time while the number of statements per procedure or function decreased, suggesting increased modularity. The major result of the study (in the *interpretation context* of the study purpose) was that the iterative enhancement technique encouraged the development of a software product that had several generally desirable aspects of system structure. A major result (in the *interpretation context* of the field of research) was that the study demonstrated the feasibility of iterative enhancement. The particular programming team and project examined qualify the *extrapolation* of the results. The *impact* of the study was an advancement in the understanding of software development approaches.

In order to improve, better understand, and manage the software development process, Baker [6] evaluated the effect of applying chief programming teams and structured programming in system development from the perspective of the user, developer, project manager, and corporate manager. The large (83 000 line) system, known as "The New York Times Project," was developed by a team of professionals organized as a chief programmer team, using structured code, top-down design, walk-throughs, and program libraries. Several benefits were identified, including reduced development time and cost, reduced time in system integration, and reduced fault detection in acceptance testing and field use. The results of the study demonstrated the feasibility of the chief programmer team concept and the accompanying methodologies in a production environment.

In order to improve their development environments, several researchers [49], [24], [2], [81], [13] have each conducted single project studies to characterize the errors and changes made during a development project. They examined the development of a moderate to large software project, done by a multiperson team, in a production environment. They analyzed the frequency and distribution of errors during development and their relationship with several factors, including module size, software complexity, developer experience, method of detection and isolation, effort for isolation and correction, phase of entrance into the system and observance, reuse of existing design and code, and role of the requirements document. Such analyses have produced fault categorization schemes and have been useful in understanding and improving a development environment.

In order to better understand and improve the use of the Ada® language, Basili *et al.* [55], [12] examined a ground-support system written in Ada to characterize the use of Ada packages from the perspective of the developer. Four professional programmers developed a project of 10 000 source lines of code. Factors such as how package use affected the ease of system modification and how to measure module change resistance were identified, as well as how these observations related to aspects of development and training. The major results were 1) several measures of Ada programs were developed, and 2) there was an indication that a lot of training will be necessary if we are to expect the facilities of Ada to be properly used.

In order to assess and improve software testing methodology, Basili and Ramsey [15], [88] characterized and evaluated the relationship between system acceptance tests and operational usage from the perspective of the developer, project manager, customer, and researcher. The execution coverage of functionally generated acceptance test cases and a sample of operational usage cases was monitored for a medium-size (10 000 line) software system developed in a production environment. The results calculated that 64 percent of the program statements were executed during system operation and that the acceptance test cases corresponded reasonably well to the operational usage. The results gave insights into the relationships among structural coverage, fault detection, system testing, and system usage.

V. Problem Areas in Experimentation

The following sections identify several problem areas of experimentation in software engineering. These areas may serve as guidelines in the performance of future studies. After mentioning some overall observations, considerations in each of the areas of experiment definition, planning, operation, and interpretation are discussed.

A. Experimentation Overall

There appears to be no "universal model" or "silver bullet" in software engineering. There are an enormous number of factors that differ across environments, in terms of desired cost/quality goals, methodology, experience, problem domain, constraints, etc. [108], [20], [5], [10], [28]. This results in every software development/maintenance environment being different. Another area of wide variation is the many-to-one (e.g., 10:1) differential in human performance [11], [43], [18]. The particular individuals examined in an empirical study can make an enormous difference. Among other considerations, these variations suggest that metrics need to be validated for a particular environment and a particular person to show that they capture what is intended [11], [21]. Thus, experimental studies should consider the potentially vast differences among environments and among people.

B. Experiment Definition

In the definition of the purpose for the experiment, the formulation of intuitive problems into precisely stated goals is a nontrivial task [17], [25]. Defining the purpose of a study often requires the articulation of what is meant by "software quality." The many interpretations and perceptions of quality [32], [39], [72] highlight the need for considering whose perspective of quality is being examined. Thus, a precise specification of the problem to be investigated is a major step toward its solution.

C. Experiment Planning

Experimental planning should have a horizon beyond a first experiment. Controlled studies may be used to focus on the effect of certain factors, while their results may be confirmed in replications [92], [99], [102], [113], [58], [57], [45], [44], [18] and/or larger case studies [5], [16]. When designing studies, consider that a combination of factors may be effective as a "critical mass," even though the particular factors may be ineffective when treated in isolation [16], [107]. Note that formal designs and the resulting statistical robustness are desirable, but we should not be driven exclusively by the achievement of statistical significance. Common sense must be maintained, which allows us, for example, to experiment just to help develop and refine hypotheses [13], [112]. Thus, the experimental planning process should include a series of experiments for exploration, verification, and application.

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

D. Experiment Operation

The collection of the required data constitutes the primary result of the study operation phase. The data must be carefully defined, validated, and communicated to ensure their consistent interpretation by all persons associated with the experiment: subjects under observation, experimenters, and literature audience [21]. There have been papers in the literature that do not define their data well enough to enable a comparison of results across many projects and environments. We have often contacted experimenters and discovered that different entities were being measured in different studies. Thus, the experimenter should be cautious about the definition, validation, and communication of data, since they play a fundamental role in the experimental process.

E. Experiment Interpretation

The appropriate presentation of results from experiments contributes to their correct interpretation. Experimental results need to be qualified by the particular samples (e.g., programmers, programs) analyzed [17]. The extrapolation of results from a particular sample must consider the representativeness of the sample to other environments [40], [114], [108], [86], [5], [28]. The visibility of the experimental results in professional forums and the open literature provides valuable feedback and constructive criticism. Thus, the presentation of experimental results should include appropriate qualification and adequate exposure to support their proper interpretation.

VI. Conclusion

Experimentation in software engineering supports the advancement of the field through an iterative learning process. The experimental process has begun to be applied in a multiplicity of environments to study a variety of software technology areas. From the studies presented, it is clear that experimentation has proven effective in providing insights and furthering our domain of knowledge about the software process and product. In fact, there is a learning process in the experimentation approach itself, as has been shown in this paper.

We have described a framework for experimentation to provide a structure for presenting previous studies. We also recommend the framework as a mechanism to facilitate the definition, planning, operation, and interpretation of past and future studies. The problem areas discussed are meant to provide some useful recommendations for the application of the experimental process in software engineering. The experimental framework cannot be used in a vacuum; the framework and the lessons learned complement one another and should be used in a synergistic fashion.

References

[1] E. N. Adams, "Optimizing preventive service of software products," *IBM J. Res. Develop.*, vol. 28, no. 1, pp. 2-14, Jan. 1984.

[2] J.-L. Albin and R. Ferreol, "Collecte et analyse de mesures de logiciel (Collection and analysis of software data)," *Technique et Science Informatiques*, vol. 1, no. 4, pp. 297-313, 1982 (Rairo ISSN 0752-4072).

[3] A. Avizienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges, "The UCLA Dedix system: A distributed testbed for multiple-version software," in *Dig. 15th Int. Symp. Fault-Tolerant Comput.*, Ann Arbor, MI, June 19-21, 1985.

[4] J. W. Bailey, "Teaching Ada: A comparison of two approaches," in *Proc. Washington Ada Symp.*, Washington, DC, 1984.

[5] J. W. Bailey and V. R. Basili, "A meta-model for software development resource expenditures," in *Proc. 5th Int. Conf. Software Eng.*, San Diego, CA, 1981, pp. 107-116.

[6] F. T. Baker, "System quality through structured programming," in *AFIPS Proc. 1972 Fall Joint Comput. Conf.*, vol. 41, 1972, pp. 339-343.

[7] V. R. Basili, *Tutorial on Models and Metrics for Software Management and Engineering.* New York: IEEE Computer Society, 1980.

[8] V. R. Basili and F. T. Baker, "Tutorial of structured programming," in *Proc. 11th IEEE COMPCON*, IEEE Cat. No. 75CH1049-6, 1975.

[9] V. R. Basili and C. Doerflinger, "Monitoring software development through dynamic variables," in *Proc. COMPSAC*, Chicago, IL, 1983.

[10] V. R. Basili and K. Freburger, "Programming measurement and estimation in the software engineering laboratory," *J. Syst. Software*, vol. 2, pp. 47-57, 1981.

[11] V. R. Basili and D. H. Hutchens, "An empirical study of a syntactic metric family," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 664-672, Nov. 1983.

[12] V. R. Basili, E. E. Katz, N. M. Panilio-Yap, C. L. Ramsey, and S. Chang, "A quantitative characterization and evaluation of a software development in Ada," *Computer*, Sept. 1985.

[13] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Commun. ACM*, vol. 27, no. 1, pp. 42-52, Jan. 1984.

[14] V. R. Basili and C. L. Ramsey, "Arrowsmith-P—A prototype expert system for software engineering management," in *Proc. Symp. Expert Systems in Government*, Mclean, VA, Oct. 1985.

[15] V. R. Basili and J. R. Ramsey, "Analyzing the test process using structural coverage," in *Proc. 8th Int. Conf. Software Eng.*, London, Aug. 28-30, 1985, pp. 306-312.

[16] V. R. Basili and R. W. Reiter, "A controlled experiment quantitatively comparing software development approaches," *IEEE Trans. Software Eng.*, vol. SE-7, May 1981.

[17] V. R. Basili and R. W. Selby, "Data collection and analysis in software research and management," *Proc. Amer. Statistical Association and Biometric Society Joint Statistical Meetings*, Philadelphia, PA, August 13-16, 1984.

[18] ——, "Comparing the effectiveness of software testing strategies," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1501, May 1985.

[19] ——, "Four applications of a software data collection and analysis methodology," in *Proc. NATO Advanced Study Institute: The Challenge of Advanced Computing Technology to System Design Methods*, Durham, U. K., July 29-Aug. 10, 1985.

[20] ——, "Calculation and use of an environment's characteristic software metric set," in *Proc. 8th Int. Conf. Software Eng.*, London, Aug. 28-30, 1985, pp. 386-393.

[21] V. R. Basili, R. W. Selby, and T. Y. Phillips, "Metric analysis and data validation across FORTRAN projects," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 652-663, Nov. 1983.

[22] V. R. Basili and A. J. Turner, "Iterative enhancement: A practical technique for software development," *IEEE Trans. Software Eng.*, vol. SE-1, Dec. 1975.

[23] ——, *SIMPL-T: A Structured Programming Language.* Geneva, IL: Paladin House, 1976.

[24] V. R. Basili and D. M. Weiss, "Evaluation of a software requirements document by analysis of change data," in *Proc. 5th Int. Conf. Software Eng.*, San Diego, CA, Mar. 9-12, 1981, pp. 314-323.

[25] ——, "A methodology for collecting valid software engineering data*," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 728-738, Nov. 1984.

[26] V. R. Basili and M. V. Zelkowitz, "Analyzing medium-scale software developments," in *Proc. 3rd Int. Conf. Software Eng.*, Atlanta, GA, May 1978, pp. 116-123.

[27] V. R. Basili, M. V. Zelkowitz, F. E. McGarry, R. W. Reiter, Jr., W. F. Truszkowski, and D. L. Weiss, "The software engineering laboratory," Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD, Rep. SEL-77-001, May 1977.

[28] B. W. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[29] B. W. Boehm, T. E. Gray, and T. Seewaldt, "Prototyping versus specifying: A multiproject experiment," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 290-303, May 1984.

[30] R. C. Bogdan and S. K. Biklen, *Qualitative Research for Education: An Introduction to Theory and Methods.* Boston, MA: Allyn and Bacon, 1982.

[31] J. Bowen, "Estimation of residual faults and testing effectiveness," in *Proc. 7th Minnowbrook Workshop Software Performance Evaluation*, Blue Mountain Lake, NY, July 24-27, 1984.

[32] T. P. Bowen, G. B. Wigle, and J. T. Tsai, "Specification of software quality attributes," Rome Air Development Center, Griffiss Air Force Base, NY, Tech. Rep. RADC-TR-85-37 (3 vols.), Feb. 1985.

[33] G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters.* New York: Wiley, 1978.

[34] F. P. Brooks, Jr., *The Mythical Man-Month.* Reading, MA: Addison-Wesley, 1975.

[35] R. E. Brooks, "Studying programmer behavior: The problem of proper methodology, *Commun. ACM*, vol. 23, no. 4, pp. 207-213, 1980.

[36] W. D. Brooks, "Software technology payoff: Some statistical evidence," *J. Syst. Software*, vol. 2, pp. 3-9, 1981.

[37] F. O. Buck, "Indicators of quality inspections," IBM Systems Products Division, Kingston, NY, Tech. Rep. 21.802, Sept. 1981.

[38] D. N. Card, F. E. McGarry, J. Page, S. Eslinger, and V. R. Basili, "The software engineering laboratory," Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD, Rep. SEL-81-104, Feb. 1982.

[39] J. P. Cavano and J. A. McCall, "A Framework for the measurement of software quality," in *Proc. Software Quality and Assurance Workshop*, San Diego, CA, Nov. 1978, pp. 133-139.

[40] W. G. Cochran, *Sampling Techniques.* New York: Wiley, 1953.

[41] W. G. Cochran and G. M. Cox, *Experimental Designs.* New York: Wiley, 1950.

[42] P. A. Currit, M. Dyer, and H. D. Mills, "Certifying the reliability of software," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 3-11, Jan. 1986.

[43] B. Curtis, "Cognitive science of programming," *6th Minnowbrook Workshop Software Performance Evaluation*, Blue Mountain Lake, NY, July 19-22, 1983.

[44] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics," *IEEE Trans. Software Eng.*, pp. 96-104, Mar. 1979.

[45] B. Curtis, S. B. Sheppard, and P. M. Milliman, "Third time charm: Stronger replication of the ability of software complexity metrics to predict programmer performance," in *Proc. 4th Int. Conf. Software Eng.*, Sept. 1979, pp. 356-360.

[46] M. Dyer, "Cleanroom software development method," IBM Federal Systems Division, Bethesda, MD, Oct. 14, 1982.

[47] M. Dyer and H. D. Mills, "Developing electronic systems with certifiable reliability," in *Proc. NATO Conf.*, Summer 1982.

[48] T. Emerson, "A discriminant metric for module cohesion," in *Proc. 7th Int. Conf. Software Eng.*, Orlando, FL, 1984, pp. 294-303.

[49] A. Endres, "An analysis of errors and their causes in systems programs," *IEEE Trans. Software Eng.*, pp. 140-149, vol. SE-1, June 1975.

[50] A. R. Feuer and E. B. Fowlkes, "Some results from an empirical study of computer software," in *Proc. 4th Int. Conf. Software Eng.*, 1979, pp. 351-355.

[51] R. W. Floyd, "Assigning meaning to programs," *Amer. Math. Soc.*, vol. 19, J. T. Schwartz, Ed., Providence, RI, 1967.

[52] K. Freburger and V. R. Basili, "The software engineering laboratory: Relationship equations," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-764, May 1979.

[53] J. D. Gannon, "An experimental evaluation of data type conventions," *Commun. ACM*, vol. 20, no. 8, pp. 584-595, 1977.

[54] J. D. Gannon and J. J. Horning, "The impact of language design on the production of reliable software," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 179-191, 1975.

[55] J. D. Gannon, E. E. Katz, and V. R. Basili, "Characterizing Ada programs: Packages," in *The Measurement of Computer Software Performance*, Los Alamos Nat. Lab., Aug. 1983.

[56] A. L. Goel, "Software reliability and estimation techniques," Rome Air Development Center, Griffiss Air Force Base, NY, Rep. RADC-TR-82-263, Oct. 1982.

[57] J. D. Gould, "Some psychological evidence on how people debug computer programs," *Int. J. Man-Machine Studies*, vol. 7, pp. 151-182, 1975.

[58] J. D. Gould and P. Drongowski, "An exploratory study of computer program debugging," *Human Factors*, vol. 16, no. 3, pp. 258-277, 1974.

[59] M. H. Halstead, *Elements of Software Science.* New York: North-Holland, 1977.

[60] W. C. Hetzel, "An experimental analysis of program verification methods," Ph.D. dissertation, Univ. North Carolina, Chapel Hill, 1976.

[61] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576-583, Oct. 1969.

[62] D. H. Hutchens and V. R. Basili, "System structure analysis: Clustering with data bindings," *IEEE Trans. Software Eng.*, vol. SE-11, Aug. 1985.

[63] S.-S. V. Hwang, "An empirical study in functional testing, structural testing, and code reading/inspection*," Dep. Comput. Sci., Univ. Maryland, College Park, Scholarly Paper 362, Dec. 1981.

[64] Z. Jelinski and P. B. Moranda, "Applications of a probability-based model to a code reading experiment," in *Proc. IEEE Symp. Comput. Software Rel.*, New York, 1973, pp. 78-81.

[65] W. L. Johnson, S. Draper, and E. Soloway, "An effective bug classification scheme must take the programmer into account," in *Proc. Workshop High-Level Debugging*, Palo Alto, CA, 1983.

[66] J. P. J. Kelly, "Specification of fault-tolerant multi-version software: Experimental studies of a design diversity approach," Ph.D. dissertation, Univ. California, Los Angeles, 1982.

[67] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 96-109, Jan. 1986.

[68] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice.* Reading, MA: Addison-Wesley, 1979.

[69] B. Littlewood, "Stochastic reliability growth: A model for fault renovation computer programs and hardware designs," *IEEE Trans. Rel.*, vol. R-30, Oct. 1981.

[70] B. Littlewood and J. L. Verrall, "A Bayesian reliability growth model for computer software," *Appl. Statist.*, vol. 22, no. 3, 1973.

[71] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308-320, Dec. 1976.

[72] J. A. McCall, P. Richards, and G. Walters, "Factors in software quality," Rome Air Development Center, Griffiss Air Force Base, NY, Tech. Rep. RADC-TR-77-369, Nov. 1977.

[73] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, "Program indentation and comprehensibility," *Commun. ACM*, vol. 26, no. 11, pp. 861-867, Nov. 1983.

[74] T. Moher and G. M. Schneider, "Methodology and experimental research in software engineering," *Int. J. Man-Machine Studies*, vol. 16, no. 1, pp. 65-87, 1982.

[75] S. A. Mulaik, *The Foundations of Factor Analysis.* New York: McGraw-Hill, 1972.

[76] J. D. Musa, "A theory of software reliability and its application," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 312-327, 1975.

[77] ——, "Software reliability measurement," *J. Syst. Software*, vol. 1, no. 3, pp. 223-241, 1980.

[78] G. L. Myers, *Composite/Structured Design.* New York: Van Nostrand Reinhold, 1978.

[79] ——, "A controlled experiment in program testing and code walkthroughs/inspections," *Commun. ACM*, pp. 760-768, Sept. 1978.

[80] J. Neter and W. Wasserman, *Applied Linear Statistical Models.* Homewood, IL: Richard D. Irwin, 1974.

[81] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment*," *J. Syst. Software*, vol. 4, pp. 289-300, 1983.

[82] D. J. Panzl, "Experience with automatic program testing," in *Proc. NBS Trends and Applications*, Nat. Bureau Standards, Gaithersburg, MD, May 28, 1981, pp. 25-28.

[83] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.

[84] ——, "Some conclusions from an experiment in software engineering techniques," in *AFIPS Proc. 1972 Fall Joint Comput. Conf.*, vol. 41, 1972, pp. 325-329.

[85] ——, "A technique for module specification with examples," *Commun. ACM*, vol. 15, May 1972.

[86] L. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Software Eng.*, vol. SE-4, July 1978.

[87] H. R. Ramsey, M. E. Atwood, and J. R. Van Doren, "Flowcharts versus program design languages: An experimental comparison," *Commun. ACM*, vol. 26, no.6, pp. 445-449, June 1983.

[88] J. Ramsey, "Structural coverage of functional testing," in *Proc. 7th Minnowbrook Workshop Software Perform. Eval.*, Blue Mountain Lake, NY, July 24-27, 1984.

[89] *Statistical Analysis System (SAS) User's Guide*, SAS Inst. Inc., Box 8000, Cary, NC 27511, 1982.

[90] H. Scheffe, *The Analysis of Variance*. New York: Wiley, 1959.

[91] "Annotated bibliography of software engineering laboratory (SEL) literature," Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD, Rep. SEL-82-006, Nov. 1982.

[92] R. W. Selby, "An empirical study comparing software testing techniques," in *Proc. 6th Minnowbrook Workshop Software Perform. Eval.*, Blue Mountain Lake, NY, July 19-22, 1983.

[93] —, "Evaluations of software technologies: Testing, CLEANROOM, and metrics," Ph.D. dissertation, Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1500, 1985.

[94] R. W. Selby, V. R. Basili, and F. T. Baker, "CLEANROOM software development: An empirical evaluation," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1415, Feb. 1985.

[95] J. G. Shanthikumar, "A statisical time dependent error occurrence rate software reliability model with imperfect debugging," in *Proc. 1981 Nat. Comput. Conf.*, June 1981.

[96] B. A. Sheil, "The psychological study of programming," *Comput. Surveys*, vol. 13, pp. 101-120, Mar. 1981.

[97] V. Y. Shen, T. J. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying error-prone software—An empirical study," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 317-324, Apr. 1985.

[98] B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop, 1980.

[99] B. Shneiderman, R. E. Mayer, D. McKay, and P. Heller, "Experimental investigations of the utility of detailed flowcharts in programming," *Commun. ACM*, vol. 20, no. 6, pp. 373-381, 1977.

[100] S. Siegel, *Nonparametric Statistics for the Behavioral Sciences*. New York: McGraw-Hill, 1955.

[101] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive strategies and looping constructs: An empirical study," *Commun. ACM*, vol. 26, no.11, pp. 853-860, Nov. 1983.

[102] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 595-609, Sept. 1984.

[103] E. Soloway, K. Ehrlich, J. Bonar, and J. Greenspan, "What do novices know about programming?" in *Directions in Human-Computer Interactions*, A. Badre and B. Shneiderman, Eds. Norwood, NJ: Ablex, 1982.

[104] W. P. Stevens, G. L. Myers, and L. L. Constantine, "Structural design, " *IBM Syst. J.*, vol. 13, no. 2, pp. 115-139, 1974.

[105] L. G. Stucki, "New directions in automated tools for improving software quality," in *Current Trends in Programming Methodology*, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977.

[106] I. Vessey and R. Weber, "Some factors affecting program repair maintenance: An empirical study," *Commun. ACM*, vol. 26, no. 2, pp. 128-134, Feb. 1983.

[107] J. Vosburgh, B. Curtis, R. Wolverton, B. Albert, H. Malec, S. Hoben, and Y. Liu, "Productivity factors and programming environments," in *Proc. 7th Int. Conf. Software Eng.*, Orlando, FL, 1984, pp. 143-152.

[108] C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, no. 1, pp. 54-73, 1977.

[109] G. Weinberg, *The Psychology of Computer Programming*. New York: Van Nostrand Rheinhold, 1971.

[110] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, pp. 446-452, July 1982.

[111] M. Weiser and J. Shertz, "Programming problem representation in novice and expert programmers," *Int. J. Man-Machine Studies*, vol. 19, pp. 391-398, 1983.

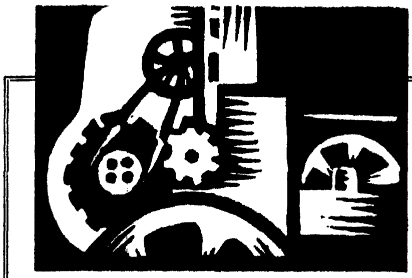
[112] D. M. Weiss and V. R. Basili, "Evaluating software development by analysis of changes: Some data from the software engineering laboratory," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 157-168, Feb. 1985.

[113] L. Weissman, "Psychological complexity of computer programs: An experimental methodology," *SIGPLAN Notices*, vol. 9, no. 6, pp. 25-36, June 1974.

[114] R. Wolverton, "The cost of developing large scale software," *IEEE Trans. Comput.*, vol. C-23, June 1974.

[115] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," Dep. Comput. Sci., Arizona State Univ., Tempe, AZ, Working Paper, 1981.

[116] J. C. Zolnowski and D. B. Simmons, "Taking the measure of program complexity," in *Proc. Nat. Comput. Conf.*, 1981, pp. 329-336.



Software Risk Management: Principles and Practices

BARRY W. BOEHM,
Defense Advanced Research Projects Agency

◆ *Identifying and dealing with risks early in development lessens long-term costs and helps prevent software disasters. It is easy to begin managing risks in your environment.*

Like many fields in their early stages, the software field has had its share of project disasters: the software equivalents of the Beauvais Cathedral, the *HMS Titanic*, and the "Galloping Gertie" Tacoma Narrows Bridge. The frequency of these software-project disasters is a serious concern: A recent survey of 600 firms indicated that 35 percent of them had at least one runaway software project.[1]

Most postmortems of these software-project disasters have indicated that their problems would have been avoided or strongly reduced if there had been an explicit early concern with identifying and resolving their high-risk elements. Frequently, these projects were swept along by a tide of optimistic enthusiasm during their early phases that caused them to miss some clear signals of high-risk issues that proved to be their downfall later.

Enthusiasm for new software capabilities is a good thing. But it must be tempered with a concern for early identification and resolution of a project's high-risk elements so people can get these resolved early and then focus their enthusiasm and energy on the positive aspects of their product.

Current approaches to the software process make it too easy for projects to make high-risk commitments that they will later regret:

♦ The sequential, document-driven waterfall process model tempts people to overpromise software capabilities in contractually binding requirements specifications before they understand their risk implications.

♦ The code-driven, evolutionary development process model tempts people to say, "Here are some neat ideas I'd like to put into this system. I'll code them up, and

Reprinted from *IEEE Software*, Vol. 8, No. 1, Jan. 1991, pp. 32–41.

if they don't fit other people's ideas, we'll just evolve things until they work." This sort of approach usually works fine in some well-supported minidomains like spreadsheet applications but, in more complex application domains, it most often creates or neglects unsalvageable high-risk elements and leads the project down the path to disaster.

At TRW and elsewhere, I have had the good fortune to observe many project managers at work firsthand and to try to understand and apply the factors that distinguished the more successful project managers from the less successful ones. Some successfully used a waterfall approach, others successfully used an evolutionary development approach, and still others successfully orchestrated complex mixtures of these and other approaches involving prototyping, simulation, commercial software, executable specifications, tiger teams, design competitions, subcontracting, and various kinds of cost-benefit analyses.

One pattern that emerged very strongly was that the successful project managers were good *risk managers*. Although they generally didn't use such terms as "risk identification," "risk assessment," "risk-management planning," or "risk monitoring," they were using a general concept of risk exposure (potential loss times the probability of loss) to guide their priorities and actions. And their projects tended to avoid pitfalls and produce good products.

The emerging discipline of software risk management is an attempt to formalize these risk-oriented correlates of success into a readily applicable set of principles and practices. Its objectives are to identify, address, and eliminate risk items before they become either threats to successful software operation or major sources of software rework.

BASIC CONCEPTS

Webster's dictionary defines "risk" as "the possibility of loss or injury." This definition can be translated into the fundamental concept of risk management: risk exposure, sometimes also called "risk impact" or "risk factor." Risk exposure is defined by the relationship

$$RE = P(UO) * L(UO)$$

where RE is the risk exposure, P(UO) is the probability of an unsatisfactory outcome and L(UO) is the loss to the parties affected if the outcome is unsatisfactory. To relate this definition to software projects, we need a definition of "unsatisfactory outcome."

Given that projects involve several classes of participants (customer, developer, user, and maintainer), each with different but highly important satisfaction criteria, it is clear that "unsatisfactory outcome" is multidimensional:

♦ For customers and developers, budget overruns and schedule slips are unsatisfactory.

♦ For users, products with the wrong functionality, user-interface shortfalls, performance shortfalls, or reliability shortfalls are unsatisfactory.

♦ For maintainers, poor-quality software is unsatisfactory.

These components of an unsatisfactory outcome provide a top-level checklist for identifying and assessing risk items.

A fundamental risk-analysis paradigm is the decision tree. Figure 1 illustrates a potentially risky situation involving the software controlling a satellite experiment. The software has been under development by the experiment team, which understands the experiment well but is inexperienced in and somewhat casual about software development. As a result, the satellite-platform manager has obtained an estimate that there is a probability P(UO) of 0.4 that the experimenters' software will have a critical error: one that will wipe out the entire experiment and cause an associated loss L(UO) of the total $20 million investment in the experiment.

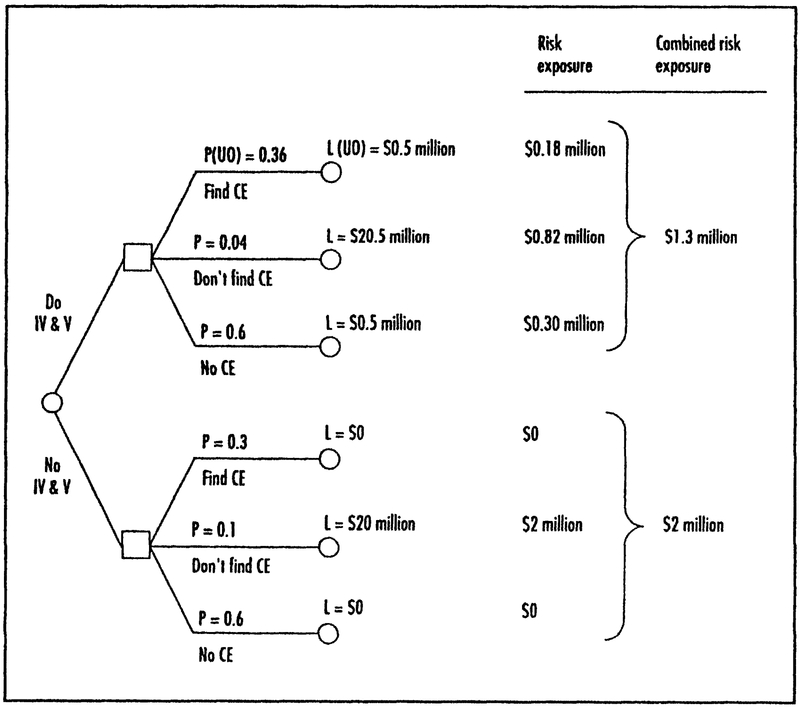


FIGURE 1. DECISION TREE FOR WHETHER TO PERFORM INDEPENDENT VALIDATION AND VERIFICATION TO ELIMINATE CRITICAL ERRORS IN A SATELLITE-EXPERIMENT PROGRAM. L(UO) IS THE LOSS ASSOCIATED WITH AN UNSATISFACTORY OUTCOME, P(UO) IS THE PROBABILITY OF THE UNSATISFACTORY OUTCOME, AND CE IS A CRITICAL ERROR.

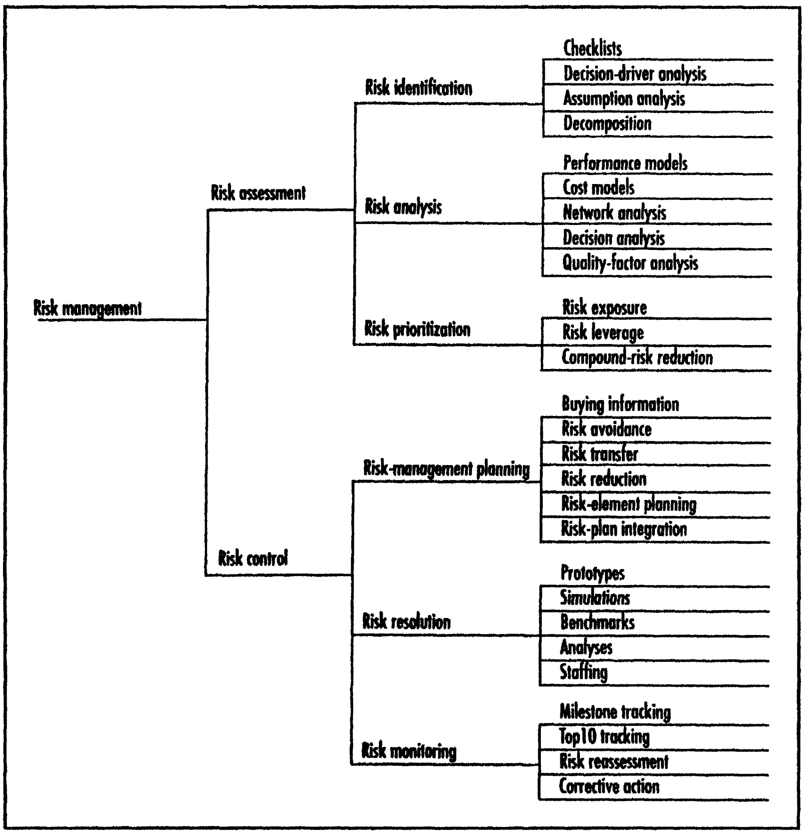


FIGURE 2. SOFTWARE RISK MANAGEMENT STEPS.

The satellite-platform manager identifies two major options for reducing the risk of losing the experiment:

♦ Convincing and helping the experiment team to apply better development methods. This incurs no additional cost and, from previous experience, the manager estimates that this will reduce the error probability P(UO) to 0.1.

♦ Hiring a contractor to independently verify and validate the software. This costs an additional $500,000; based on the results of similar IV&V efforts, the manager estimates that this will reduce the error probability P(UO) to 0.04.

The decision tree in Figure 1 then shows, for each of the two major decision options, the possible outcomes in terms of the critical error existing or being found and eliminated, their probabilities, the losses associated with each outcome, the risk exposure associated with each outcome, and the total risk exposure (or expected loss) associated with each decision option. In this case, the total risk exposure associated with the experiment-team option is only $2 million. For the IV&V option, the total risk exposure is only $1.3 million, so it represents the more attractive option.

Besides providing individual solutions for risk-management situations, the decision tree also provides a framework for analyzing the sensitivity of preferred solutions to the risk-exposure parameters. Thus, for example, the experiment-team option would be preferred if the loss due to a critical error were less than $13 million, if the experiment team could reduce its critical-error probability to less than 0.065, if the IV&V team cost more than $1.2 million, if the IV&V team could not reduce the probability of critical error to less than 0.075, or if there were various partial combinations of these possibilities.

This sort of sensitivity analysis helps deal with many situations in which probabilities and losses cannot be estimated well enough to perform a precise analysis. The risk-exposure framework also supports some even more approximate but still very useful approaches, like range estimation and scale-of-10 estimation.

RISK MANAGMENT

As Figure 2 shows, the practice of risk management involves two primary steps each with three subsidiary steps.

The first primary step, risk assessment, involves risk identification, risk analysis, and risk prioritization:

♦ Risk identification produces lists of the project-specific risk items likely to compromise a project's success. Typical risk-identification techniques include checklists, examination of decision drivers, comparison with experience (assumption analysis), and decomposition.

♦ Risk analysis assesses the loss probability and loss magnitude for each identified risk item, and it assesses compound risks in risk-item interactions. Typical techniques include performance models, cost models, network analysis, statistical decision analysis, and quality-factor (like reliability, availability, and security) analysis.

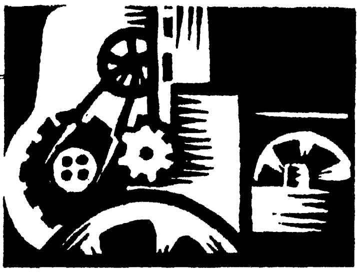
♦ Risk prioritization produces a ranked ordering of the risk items identified and analyzed. Typical techniques include risk-exposure analysis, risk-reduction leverage analysis (particularly involving cost-benefit analysis), and Delphi or group-consensus techniques.

The second primary step, risk control, involves risk-management planning, risk resolution, and risk monitoring:

♦ Risk-management planning helps prepare you to address each risk item (for example, via information buying, risk avoidance, risk transfer, or risk reduction), including the coordination of the individual risk-item plans with each other and with the overall project plan. Typical techniques include checklists of risk-resolution techniques, cost-benefit analysis, and standard risk-management plan outlines, forms, and elements.

♦ Risk resolution produces a situation in which the risk items are eliminated or otherwise resolved (for example, risk avoidance via relaxation of requirements). Typical techniques include prototypes, simulations, benchmarks, mission analyses, key-personnel agreements, design-to-cost approaches, and incremental development.

♦ Risk monitoring involves tracking the project's progress toward resolving its risk items and taking corrective action where appropriate. Typical techniques include milestone tracking and a top-10 risk-item list that is highlighted at each



weekly, monthly, or milestone project review and followed up appropriately with reassessment of the risk item or corrective action.

In addition, risk management provides an improved way to address and organize the life cycle. Risk-driven approaches, like the spiral model of the software process,[2] avoid many of the difficulties encountered with previous process models like the waterfall model and the evolutionary development model. Such risk-driven approaches also show how and where to incorporate new software technologies like rapid prototyping, fourth-generation languages, and commercial software products into the life cycle.

SIX STEPS

Figure 2 summarized the major steps and techniques involved in software risk management. This overview article covers four significant subsets of risk-management techniques: risk-identification checklists, risk prioritization, risk-management planning, and risk monitoring. Other techniques have been covered elsewhere.[3,4]

Risk-identification checklists. Table 1 shows a top-level risk-identification checklist with the top 10 primary sources of risk on software projects, based on a survey of several experienced project managers. Managers and system engineers can use the checklist on projects to help identify and resolve the most serious risk items on the project. It also provides a corresponding set of risk-management techniques that have been most successful to date in avoiding or resolving the source of risk.

If you focus on item 2 of the top-10 list in Table 1 (unrealistic schedules and budgets), you can then move on to an example of a next-level checklist: the risk-probability table in Table 2 for assessing the probability that a project will overrun its budget. Table 2 is one of several such checklists in an excellent US Air Force handbook[5] on software risk abatement.

Using the checklist, you can rate a project's status for the individual attributes associated with its requirements, personnel, reusable software, tools, and support environment (in Table 2, the environment's availability or the risk that the environment will not be available when needed). These ratings will support a probability-range estimation of whether the project has a relatively low (0.0 to 0.3), medium (0.4 to 0.6), or high (0.7 to 1.0) probability of overrunning its budget.

Most of the critical risk items in the checklist have to do with shortfalls in domain understanding and in properly scoping the job to be done — areas that are generally underemphasized in computer-science literature and education. Recent

TABLE 1.
TOP 10 SOFTWARE RISK ITEMS.

Risk item	Risk-management technique
Personnel shortfalls	Staffing with top talent, job matching, team building, key personnel agreements, cross training.
Unrealistic schedules and budgets	Detailed multisource cost and schedule estimation, design to cost, incremental development, software reuse, requirements scrubbing.
Developing the wrong functions and properties	Organization analysis, mission analysis, operations-concept formulation, user surveys and user participation, prototyping, early users' manuals, off-nominal performance analysis, quality-factor analysis.
Developing the wrong user interface	Prototyping, scenarios, task analysis, user participation.
Gold-plating	Requirements scrubbing, prototyping, cost-benefit analysis, designing to cost.
Continuing stream of requirements changes	High change threshold, information hiding, incremental development (deferring changes to later increments).
Shortfalls in externally furnished components	Benchmarking, inspections, reference checking, compatibility analysis.
Shortfalls in externally performed tasks	Reference checking, preaward audits, award-fee contracts, competitive design or prototyping, team-building.
Real-time performance shortfalls	Simulation, benchmarking, modeling, prototyping, instrumentation, tuning.
Straining computer-science capabilities	Technical analysis, cost-benefit analysis, prototyping, reference checking.

TABLE 2.
QUANTIFICATION OF PROBABILITY AND IMPACT FOR COST FAILURE.

	Probability		
Cost drivers	Improbable (0.0-0.3)	Probable (0.4-0.6)	Frequent (0.7-1.0)
Requirements			
Size	Small, noncomplex, or easily decomposed	Medium to moderate complexity, decomposable	Large, highly complex, or not decomposable
Resource constraints	Little or no hardware-imposed constraints	Some hardware-imposed constraints	Significant hardware-imposed constraints
Application	Nonreal-time, little system interdependency	Embedded, some system interdependencies	Real-time, embedded, strong interdependency
Technology	Mature, existent, in-house experience	Existent, some in-house experience	New or new application, little experience
Requirements stability	Little or no change to established baseline	Some change in baseline expected	Rapidly changing, or no baseline
Personnel			
Availability	In place, little turnover expected	Available, some turnover expected	Not available, high turnover expected
Mix	Good mix of software disciplines	Some disciplines inappropriately represented	Some disciplines not represented
Experience	High experience ratio	Average experience ratio	Low experience ratio
Management environment	Strong personnel management approach	Good personnel management approach	Weak personnel management approach
Reusable software			
Availability	Compatible with need dates	Delivery dates in question	Incompatible with need dates
Modifications	Little or no change	Some change	Extensive changes
Language	Compatible with system and maintenance requirements	Partial compatibility with requirements	Incompatible with system or maintenance requirements
Rights	Compatible with maintenance and competition requirements	Partial compatibility with maintenance, some competition	Incompatible with maintenance concept, noncompetitive
Certification	Verified performance, application compatible	Some application-compatible test data available	Unverified, little test data available
Tools and environment			
Facilities	Little or no modification	Some modifications, existent	Major modifications, nonexistent
Availability	In place, meets need dates	Some compatibility with need dates	Nonexistent, does not meet need dates
Rights	Compatible with maintenance and development plans	Partial compatibility with maintenance and development plans	Incompatible with maintenance and development plans
Configuration management	Fully controlled	Some controls	No controls
	Impact		
	Sufficient financial resources	Some shortage of financial resources, possible overrun	Significant financial shortages, budget overrun likely

initiatives, like the Software Engineering Institute's masters curriculum in software engineering, are providing better coverage in these areas. The SEI is also initiating a major new program in software risk management.

Risk analysis and prioritization. After using all the various risk-identification checklists, plus the other risk-identification techniques in decision-driver analysis, assumption analysis, and decomposition, one very real risk is that the project will identify so many risk items that the project could spend years just investigating them. This is where risk prioritization and its associated risk-analysis activities become essential.

The most effective technique for risk prioritization involves the risk-exposure quantity described earlier. It lets you rank the risk items identified and determine which are most important to address.

One difficulty with the risk-exposure

TABLE 3.
RISK EXPOSURE FACTORS FOR SATELLITE EXPERIMENT SOFTWARE.

Unsatisfactory outcome	Probability of unsatisfactory outcome	Loss caused by unsatisfactory outcome	Risk exposure
A. Software error kills experiment	3-5	10	30-50
B. Software error loses key data	3-5	8	24-40
C. Fault-tolerant features cause unacceptable performance	4-8	7	28-56
D. Monitoring software reports unsafe condition as safe	5	9	45
E. Monitoring software reports safe condition as unsafe	5	3	15
F. Hardware delay causes schedule overrun	6	4	24
G. Data-reduction software errors cause extra work	8	1	8
H. Poor user interface causes inefficient operation	6	5	30
I. Processor memory insufficient	1	7	7
J. Database-management software loses derived data	2	2	4

quantity, as with most other decision-analysis quantities, is the problem of making accurate input estimates of the probability and loss associated with an unsatisfactory outcome. Checklists like that in Table 2 provide some help in assessing the probability of occurrence of a given risk item, but it is clear from Table 2 that its probability ranges do not support precise probability estimation.

Full risk-analysis efforts involving prototyping, benchmarking, and simulation generally provide better probability and loss estimates, but they may be more expensive and time-consuming than the situation warrants. Other techniques, like betting analogies and group-consensus techniques, can improve risk-probability estimation, but for risk prioritization you can often take a simpler course: assessing the risk probabilities and losses on a relative scale of 0 to 10.

Table 3 and Figure 3 illustrate this risk-prioritization process by using some potential risk items from the satellite-experiment project as examples. Table 3 summarizes several unsatisfactory outcomes with their corresponding ratings for P(UO), L(UO), and their resulting risk-exposure estimates. Figure 3 plots each unsatisfactory outcome with respect to a set of constant risk-exposure contours.

Three key points emerge from Table 3 and Figure 3:

♦ Projects often focus on factors having either a high P(UO) or a high L(UO), but these may not be the key factors with a high risk-exposure combination. One of the highest P(UO)s comes from item G (data-reduction errors), but the fact that these errors are recoverable and not mission-critical leads to a low loss factor and a resulting low RE of 7. Similarly, item I (insufficient memory) has a high potential loss, but its low probability leads to a low RE of 7. On the other hand, a relatively low-profile item like item H (user-interface shortfalls) becomes a relatively high-priority risk item because its combination of moderately high probability and loss factors yield a RE of 30.

♦ The RE quantities also provide a basis for prioritizing verification and vali-

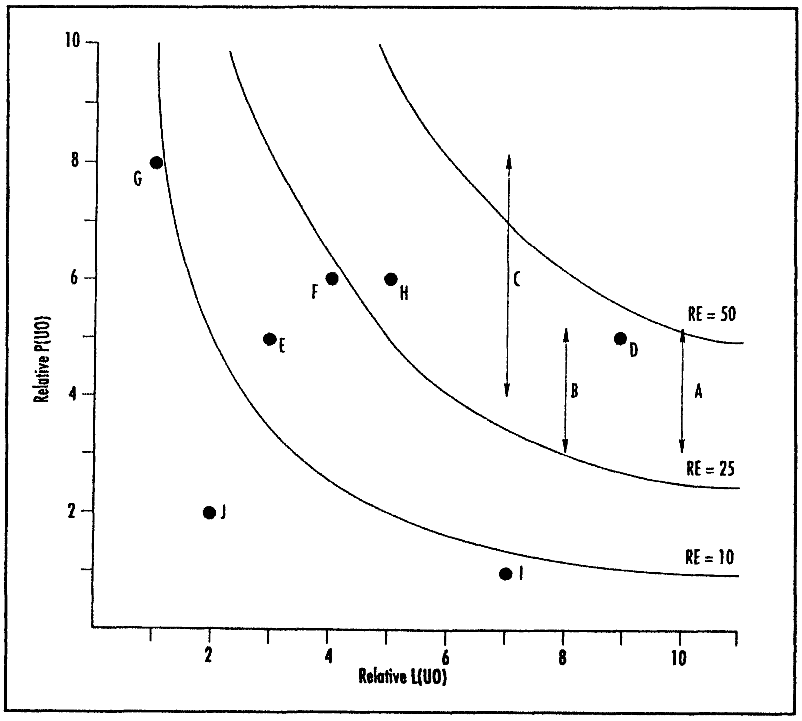


FIGURE 3. RISK-EXPOSURE FACTORS AND CONTOURS FOR THE SATELLITE-EXPERIMENT SOFTWARE. RE IS THE RISK EXPOSURE, P(UO) THE PROBABILITY OF AN UNSATISFACTORY OUTCOME, AND L(UO) THE LOSS ASSOCIATED WITH THAT UNSATISFACTORY OUTCOME. THE GRAPH POINTS MAP THE ITEMS FROM TABLE 3 WHOSE RISK EXPOSURE ARE BEING ASSESSED.

1. Objectives (the "why)
- ◆ Determine, reduce level of risk of the software fault-tolerance features causing unacceptable performance.
- ◆ Create a description of and a development plan for a set of low-risk fault-tolerance features.

2. Deliverables and milestones (the "what" and "when").
   - ◆ By Week 3.
     1. Evaluation of fault-tolerance options
     2. Assessment of reusable components
     3. Draft workload characterization
     4. Evaluation plan for prototype exercise
     5. Description of prototype
   - ◆ By Week 7.
     6. Operational prototype with key fault-tolerance features.
     7. Workload simulation
     8. Instrumentation and data reduction capabilities.
     9. Draft description, plan for fault-tolerance features.
   - ◆ By Week 10
     10. Evaulation and iteration of prototype
     11. Revised description, plan for fault-tolerance features

3. Responsiblities (the "who" and "where")
   - ◆ System engineer: G.Smith
     Tasks 1, 3, 4, 9, 11. Support of tasks 5, 10
   - ◆ Lead programmer: C.Lee
     Tasks 5, 6, 7, 10. Support of tasks 1, 3
   - ◆ Programmer: J.Wilson
     Tasks 2, 8. Support of tasks 5, 6, 7, 10

4. Approach (the "how")
   - ◆ Design-to-schedule prototyping effort
   - ◆ Driven by hypotheses about fault-tolerance-performance effects
   - ◆ Use real-time operating system, add prototype fault-tolerance features
   - ◆ Evaluate performance with respect to representative workload
   - ◆ Refine prototype based on results observed

5. Resources (the "how much")

   $60K — full-time system engineer, lead programmer, programmer
   (10 weeks)*(3 staff)*$2k/staff-week)
   $0 — three dedicated workstations (from project pool)
   $0 — two target processors (from project pool)
   $0 — one test coprocessor (from project pool)
   $10K — contingencies
   $70K — total

FIGURE 4. RISK-MANAGEMENT PLAN FOR FAULT-TOLERANCE PROTOTYPING.

dation and related test activities by giving each error class a significance weight. Frequently, all errors are treated with equal weight, putting too much testing effort into finding relatively trivial errors.

◆ There is often a good deal of uncertainty in estimating the probability or loss associated with an unsatisfactory outcome. (The assessments are frequently subjective and are often the product of surveying several domain experts.) The amount of uncertainty is itself a major source of risk, which needs to be reduced as early as possible. The primary example in Table 3 and Figure 3 is the uncertainty in item C about whether the fault-tolerance features are going to cause an unacceptable degradation in real-time performance. If P(UO) is rated at 4, this item has only a moderate RE of 28, but if P(UO) is 8, the RE has a top-priority rating of 56.

One of the best ways to reduce this source of risk is to buy information about the actual situation. For the issue of fault tolerance versus performance, a good way to buy information is to invest in a prototype, to better understand the performance effects of the various fault-tolerance features.

Risk-management planning. Once you determine a project's major risk items and their relative priorities, you need to establish a set of risk-control functions to bring the risk items under control. The first step in this process is to develop a set of risk-management plans that lay out the activities necessary to bring the risk items under control.

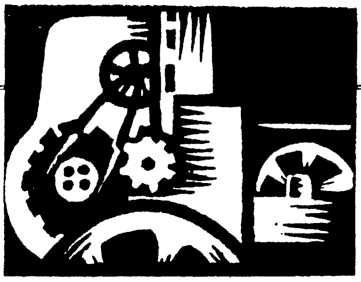
One aid in doing this is the top-10 checklist in Figure 3 that identifies the most successful risk-management techniques for the most common risk items. As an example, item 9 (real-time performance shortfalls) in Table 1 covers the uncertainty in performance effect of the fault-tolerance features. The corresponding risk-management techniques include simulation, benchmarking, modeling, prototyping, instrumentation, and tuning. Assume, for example, that a prototype of representative safety features is the most cost-effective way to determine and reduce their effects on system performance.

The next step in risk-management planning is to develop risk-management plans for each risk item. Figure 4 shows the plan for prototyping the fault-tolerance features and determining their effects on performance. The plan is organized around a standard format for software plans, oriented around answering the standard questions of why, what, when, who, where, how, and how much. This plan organization lets the plans be concise (fitting on one page), action-oriented, easy to understand, and easy to monitor.

The final step in risk-management planning is to integrate the risk-management plans for each risk item with each other and with the overall project plan. Each of the other high-priority or uncertain risk items will have a risk-management plan; it may turn out, for example, that the fault-tolerance features prototyped for this risk item could also be useful as part of the strategy to reduce the uncertainty in items A and B (software errors killing the experiment and losing experiment-critical data). Also, for the overall project plan, the need for a 10-week prototype-development and -exercise period must be factored into the overall schedule, to keep the overall schedule realistic.

Risk resolution and monitoring. Once you have established a good set of risk-management plans, the risk-resolution process consists of implementing whatever prototypes, simulations, benchmarks, surveys, or other risk-reduction techniques are called for in the plans. Risk monitoring ensures that this is a closed-loop process by tracking risk-reduction progress and applying whatever corrective action is necessary to keep the risk-resolution process on track.

Risk management provides managers with a very effective technique for keeping on top of projects under their control: *Project top-10 risk-item tracking.* This technique concentrates management atten-



tion on the high-risk, high-leverage, critical success factors rather than swamping management reviews with lots of low-priority detail. As a manager, I have found that this type of risk-item-oriented review saves a lot of time, reduces management surprises, and gets you focused on the high-leverage issues where you can make a difference as a manager.

Top-10 risk-item tracking involves the following steps:

♦ Ranking the project's most significant risk items.

♦ Establishing a regular schedule for higher management reviews of the project's progress. The review should be chaired by the equivalent of the project manager's boss. For large projects (more than 20 people), the reviews should be held monthly. In the project itself, the project manager would review them more frequently.

♦ Beginning each project-review meeting with a summary of progress on the top 10 risk items. (The number could be seven or 12 without loss of intent.) The summary should include each risk item's current top-10 ranking, its rank at the previous review, how often it has been on the top-10 list, and a summary of progress in resolving the risk item since the previous review.

♦ Focusing the project-review meeting on dealing with any problems in resolving the risk items.

Table 4 shows how a top-10 list could have worked for the satellite-experiment project, as of month 3 of the project. The project's top risk item in month 3 is a critical staffing problem. Highlighting it in the monthly review meeting would stimulate a discussion by the project team and the boss of the staffing options: Make the unavailable key person available, reshuffle project personnel, or look for new people within or outside the organization. This should result in an assignment of action items to follow through on the options chosen, including possible actions by the project manager's boss.

The number 2 risk item in Table 4, target hardware delivery delays, is also one for which the project manager's boss may be able to expedite a solution — by cutting through corporate-procurement red tape, for example, or by escalating vendor-delay issues with the vendor's higher management.

As Table 4 shows, some risk items are moving down in priority or going off the list, while others are escalating or coming onto the list. The ones moving down the list — like the design-verification and -validation staffing, fault-tolerance prototyping, and user-interface prototyping — still need to be monitored but frequently do not need special management action. The ones moving up or onto the list — like the data-bus design changes and the testbed-interface definitions — are generally the ones needing higher management attention to help get them

TABLE 4.
PROJECT TOP-10 RISK ITEM LIST FOR SATELLITE EXPERIMENT SOFTWARE.

Risk item	Monthly ranking			Risk-resolution progress
	This	Last	No. of months	
Replacing sensor-control software developer	1	4	2	Top replacement candidate unavailable
Target hardware delivery delays	2	5	2	Procurement procedural delays
Sensor data formats undefined	3	3	3	Action items to software, sensor teams; due next month
Staffing of design V&V team	4	2	3	Key reviewers committed; need fault-tolerance reviewer
Software fault-tolerance may compromise performance	5	1	3	Fault-tolerance prototype successful
Accommodate changes in data bus design	6	—	1	Meeting scheduled with data-bus designers
Test-bed interface definitions	7	8	3	Some delays in action items; review meeting scheduled
User interface uncertainties	8	6	3	User interface prototype successful
TBDs in experiment operational concept	—	7	3	TBDs resolved
Uncertainties in reusable monitoring software	—	9	3	Required design changes small, successfully made

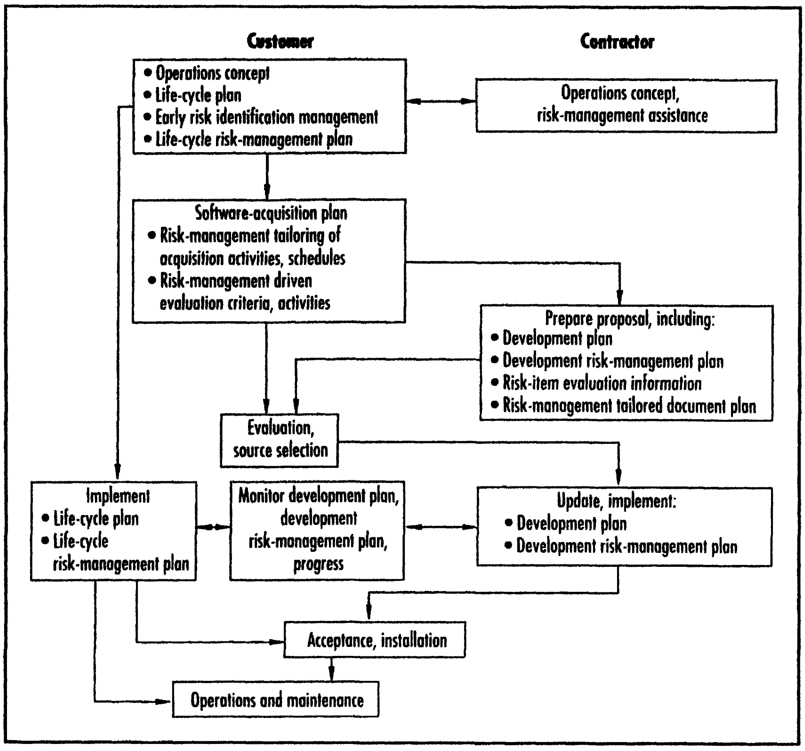

Customer
Contractor
• Operations concept
• Life-cycle plan
• Early risk identification management
• Life-cycle risk-management plan
Operations concept, risk-management assistance
Software-acquisition plan
• Risk-management tailoring of acquisition activities, schedules
• Risk-management driven evaluation criteria, activities
Prepare proposal, including:
• Development plan
• Development risk-management plan
• Risk-item evaluation information
• Risk-management tailored document plan
Evaluation, source selection
Implement
• Life-cycle plan
• Life-cycle risk-management plan
Monitor development plan, development risk-management plan, progress
Update, implement:
• Development plan
• Development risk-management plan
Acceptance, installation
Operations and maintenance

FIGURE 5. FRAMEWORK FOR LIFE-CYCLE RISK MANAGEMENT.

resolved quickly.

As this example shows, the top-10 risk-item list is a very effective way to focus higher management attention onto the project's critical success factors. It also uses management's time very efficiently, unlike typical monthly reviews, which spend most of their time on things the higher manager can't do anything about. Also, if the higher manager surfaces an additional concern, it is easy to add it to the top-10 risk item list to be highlighted in future reviews.

IMPLEMENTING RISK MANAGEMENT

Implementing risk management involves inserting the risk-management principles and practices into your existing life-cycle management practices. Full implementation of risk management involves the use of risk-driven software-process models like the spiral model, where risk considerations determine the overall sequence of life-cycle activities, the use of prototypes and other risk-resolution techniques, and the degree of detail of plans and specifications. However, the best implementation strategy is an incremental one, which lets an organization's culture adjust gradually to risk-oriented management practices and risk-driven process models.

A good way to begin is to establish a top-l0 risk-item tracking process. It is easy and inexpensive to implement, provides early improvements, and begins establishing a familiarity with the other risk-management principles and practices. Another good way to gain familiarity is via books like my recent tutorial on risk management,[3] which contains the Air Force risk-abatement pamphlet[5] and other useful articles, and Robert Charette's recent good book on risk management.[4]

An effective next step is to identify an appropriate initial project in which to implement a top-level life-cycle risk-management plan. Once the organization has accumulated some risk-management experience on this initial project, successive steps can deepen the sophistication of the risk-management techniques and broaden their application to wider classes of projects.

Figure 5 provides a scheme for implementing a top-level life-cycle risk-management plan. It is presented in the context of a contractual software acquisition, but you can tailor it to the needs of an internal development organization as well.

You can organize the life-cycle risk-management plan as an elaboration of the "why, what, when, who, where, how, how much" framework of Figure 4. While this plan is primarily the customer's responsibility, it is very useful to involve the developer community in its preparation as well.

Such a plan addresses not only the development risks that have been the prime topic of this article but also operations and maintenance risks. These include such items as staffing and training of maintenance personnel, discontinuities in the switch from the old to the new system, undefined responsibilities for operations and maintenance facilities and functions, and insufficient budget for planned life-cycle improvements or for corrective, adaptive, and perfective maintenance.

Figure 5 also shows the importance of proposed developer risk-management plans in competitive source evaluation and selection. Emphasizing the realism and effectiveness of a bidder's risk-management plan increases the probability that the customer will select a bidder that clearly understands the project's critical success factors and that has established a development approach that satisfactorily addresses them. (If the developer is a noncompetitive internal organization, it is equally important for the internal customer to require and review a developer risk-management plan.)

The most important thing for a project to do is to get focused on its critical success factors.

For various reasons, including the influence of previous document-driven management guidelines, projects get focused on activities that are not critical for their success. These frequently include writing boilerplate documents, exploring intriguing but peripheral technical issues, playing politics, and trying to sell the "ultimate" system.

In the process, critical success factors get neglected, the project fails, and nobody wins.

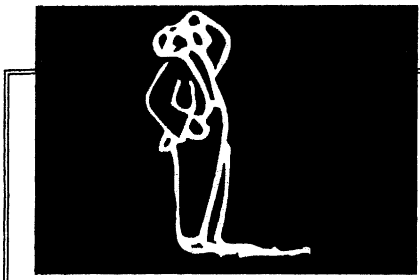
The key contribution of software risk management is to create this focus on critical success factors — and to provide the techniques that let the project deal with them. The risk-assessment and risk-control techniques presented here provide the foundation layer of capabilities needed to implement the risk-oriented approach.

However, risk management is not a cookbook approach. To handle all the complex people-oriented and technology-driven success factors in projects, a great measure of human judgement is required.

Good people, with good skills and good judgment, are what make projects work. Risk management can provide you with some of the skills, an emphasis on getting good people, and a good conceptual framework for sharpening your judgement. I hope you can find these useful on your next project. ♦

REFERENCES

1. J. Rothfeder, "It's Late, Costly, and Incompetent — But Try Firing a Computer System," *Business Week*, Nov. 7, 1988, pp. 164-165.
2. B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, May 1988, pp. 61-72.
3. B.W. Boehm, *Software Risk Management*, CS Press, Los Alamitos, Calif., 1989.
4. R.N. Charette, *Software Engineering Risk Analysis and Management*, McGraw-Hill, New York, 1989.
5. "Software Risk Abatement," AFSC/AFLC pamphlet 800-45. US Air Force Systems Command, Andrews AFB, Md., 1988.



Achieving Higher SEI Levels

MICHAEL K. DASKALANTONAKIS, Motorola

◆ *Two years or more can pass between formal SEI assessments. An organization seeking to monitor its progress to a higher SEI level needs a method for internally conducting incremental assessments. The author provides one that has proven successful at Motorola.*

Many organizations have turned to the Software Engineering Institute's Capability Maturity Model to improve their software-engineering processes by setting goals to achieve higher SEI levels. This has created the need for an instrument and a process that can be used to evaluate an organization's current status relative to these goals.[1-3] At Motorola, we have developed a method for assessing progress to higher SEI levels that lets engineers and managers evaluate an organization's current status relative to the CMM and identify weak areas for immediate attention and improvement.[4] This method serves as an effective means to ensure continuous process improvement as well as grassroots participation and support in achieving higher maturity levels.

This progress-assessment process is not intended as a replacement for any formal assessment instruments developed by the SEI, but rather as an internal tool to help organizations prepare for a formal SEI assessment. Although I provide examples in terms of CMM version 1.1, both the self-evaluation instrument and the progress-assessment process are generic enough for use with any (similar) later version of the SEI CMM by updating the worksheets and charts used.

We began using the SEI Progress-Assessment method within Motorola's Cellular Infrastructure Group — an organization of more than 1,000 software engineers working on several projects and products for the cellular com-

Reprinted from *IEEE Software*, Vol. 11, No. 4, July 1994, pp. 17–24.

Score	Key activity evaluation dimensions		
	Approach	Deployment	Results
Poor (0)	• No management recognition of need • No organizational ability • No organizational commitment • Practice not evident	• No part of the organization uses the practice • No part of the organization shows interest	• Ineffective
Weak (2)	• Management has begun to recognize the need • Support items for the practice start to be created • A few parts of organization are able to implement the practice	• Fragmented use • Inconsistent use • Deployed in some parts of the organization • Limited monitoring/verification of use	• Spotty results • Inconsistent results • Some evidence of effectiveness for some parts of the organization
Fair (4)	• Wide but not complete commitment by management • Road map for practice implementation defined • Several supporting items for the practice in place	• Less fragmented use • Some consistency in use • Deployed in some major parts of the organization • Monitoring/verification of use for several parts of the organization	• Consistent and positive results for several parts of the organization • Inconsistent results for other parts of the organization
Marginally qualified (6)	• Some management commitment; some management becomes proactive • Practice implementation well under way across parts of the organization • Supporting items in place	• Deployed in some parts of the organization • Mostly consistent use across many parts of the organization • Monitoring/verification of use for many parts of the organization	• Positive measurable results in most parts of the organization • Consistently positive results over time across many parts of the organization
Qualified (8)	• Total management commitment • Majority of management is proactive • Practice established as an integral part of the process • Supporting items encourage and facilitate the use of the practice	• Deployed in almost all parts of the organization • Consistent use across almost all parts of the organization • Monitoring/verification of use for almost all parts of the organization	• Positive measurable results in almost all parts of the organization • Consistently positive results over time across almost all parts of the organization
Outstanding (10)	• Management provides zealous leadership and commitment • Organizational excellence in the practice recognized even outside the company	• Pervasive and consistent deployment across all parts of the organization • Consistent use over time across all parts of the organization • Monitoring/verification for all parts of the organization	• Requirements exceeded • Consistently world-class results • Counsel sought by others

Figure 1. *Guidelines to rate CMM key activities in CMM version 1.1 or any later SEI CMM version. They were developed by modifying the Quality System Review scoring matrix guidelines to ensure that they address the spirit and themes considered in the CMM. All three evaluation dimensions included in this scoring matrix are equally weighted. You determine the score for a key activity by examining all three evaluation dimensions and their scoring guidelines simultaneously. An odd-numbered score is possible if some of, but not all, the criteria for the next higher level have been met.*

munications business — in the second quarter of 1992. A year later, our organization was found to have achieved SEI level 2, the next higher SEI maturity level. This was primarily the result of strong senior-management support, backed by allocation of at least 10 percent of the progress-assessment participants' efforts within a given quarter, and engineer/manager actions taken to implement the process-improvement action plans. These action plans were generated and driven through the assessment method described here.

At Motorola, we found the progress-assessment method offers several benefits. It empowers engineers and managers working within a product group to conduct a self-evaluation relative to an SEI level and create their own list of findings and action plans. This ensures grass-roots involvement in the process and institutionalization of improvement. The process facilitates communication among those involved in this assessment and ensures that important information regarding processes and tools used within the product group is disseminated at the assessment meeting and at subsequent meetings. The process educates engineers and managers — the practitioners — regarding the key process areas and practices listed in the CMM. This increases their understanding of topics in which they may not have been involved in the past, such as software configuration management or software subcontractor management. This also increases the capability of the practitioners in terms of the software-engineering process, methods, tools, and technology. Finally, the progress-assessment process continuously prepares an organization for the next formal SEI assessment.

Some critics of the assessment instrument within Motorola's CIG have said that it focuses primarily on the key activities listed in the CMM without adequately covering other key practices (also called themes) such as the commitment and ability to perform. Responding to input from the CIG's Process Management Working Group, I decided to formally score and track only the key activities, while ensuring that the scoring guidelines used for determining the key activities' scores account for the additional practices listed in the CMM. For example, to achieve a rating of Marginally Qualified, the key-activity scoring guidelines in Figure 1 require that an organization show the existence of management commitment, have supporting items in place, and monitor and verify use. Also, the progress-assessment process specification requires that findings regarding these additional practices and their associated actions be identified and used as part of an SEI Progress Assessment. This ensures the necessary coverage of these practices.

ASSESSMENT INSTRUMENT

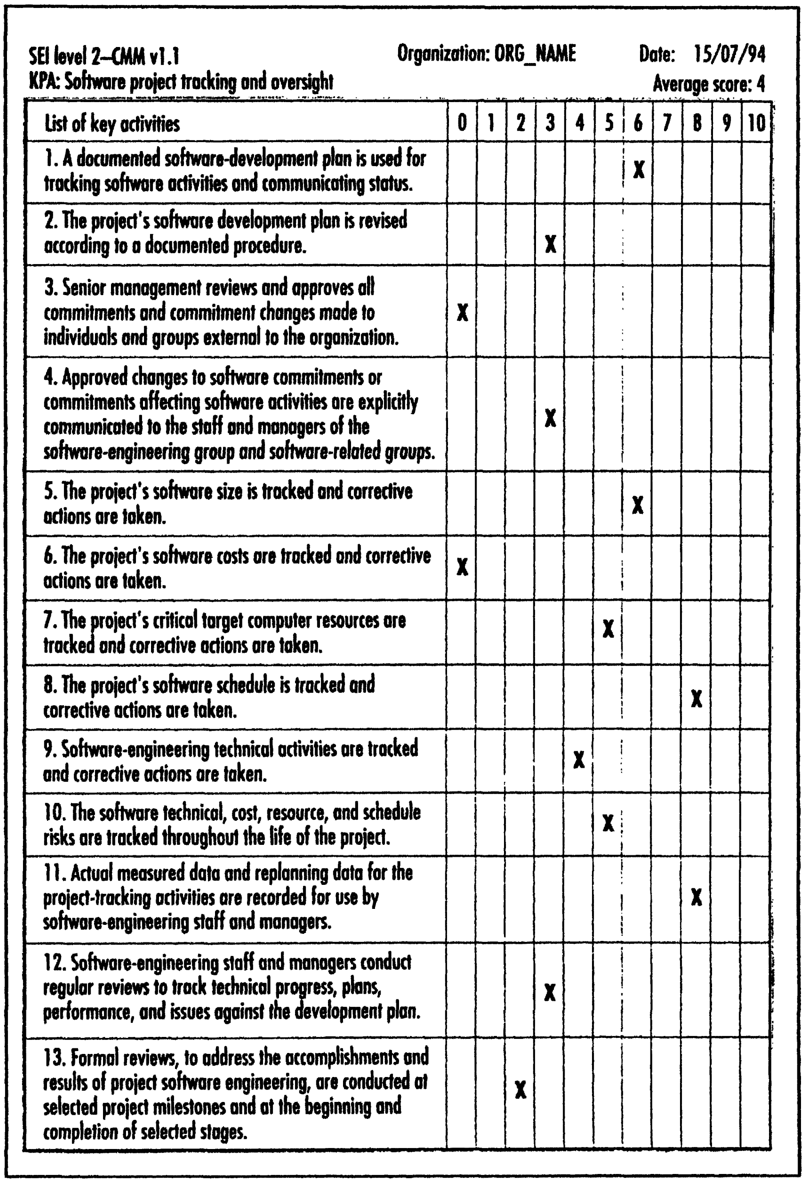
Each SEI level has several associated key process areas. The progress-assessment instrument lets you determine the scores associated with the SEI level your organization is trying to achieve. Each key process area contains several key activities. We created scoring guidelines for measuring how well an organization implements a specific key activity, basing them on several common CMM themes identified by Mark Paulk.[1]

- Commitment to perform
- Ability to perform
- Activities performed
- Monitoring implementation
- Verifying implementation

I then expanded and grouped these themes under three primary evaluation dimensions and developed criteria for evaluating them:

- *Approach.* Criteria here are the organization's commitment to and management's support for the practice, as well as the organization's ability to implement the practice.
- *Deployment.* The breadth and consistency of practice implementation across project areas are the key criteria here.
- *Results.* Criteria here are the breadth and consistency of positive results over time and across project areas.

Scoring. I used the evaluation dimensions and criteria to create guidelines for determining an integer score of 0-10 for each key activity, as Figure 1 shows. Although the guidelines are generic, the assessor can easily use them to determine the score of each specific key activity. This is simpler than having



SEI level 2—CMM v1.1 Organization: ORG_NAME Date: 15/07/94
KPA: Software project tracking and oversight Average score: 4

List of key activities	0	1	2	3	4	5	6	7	8	9	10
1. A documented software-development plan is used for tracking software activities and communicating status.							X				
2. The project's software development plan is revised according to a documented procedure.				X							
3. Senior management reviews and approves all commitments and commitment changes made to individuals and groups external to the organization.	X										
4. Approved changes to software commitments or commitments affecting software activities are explicitly communicated to the staff and managers of the software-engineering group and software-related groups.				X							
5. The project's software size is tracked and corrective actions are taken.							X				
6. The project's software costs are tracked and corrective actions are taken.	X										
7. The project's critical target computer resources are tracked and corrective actions are taken.						X					
8. The project's software schedule is tracked and corrective actions are taken.									X		
9. Software-engineering technical activities are tracked and corrective actions are taken.					X						
10. The software technical, cost, resource, and schedule risks are tracked throughout the life of the project.						X					
11. Actual measured data and replanning data for the project-tracking activities are recorded for use by software-engineering staff and managers.									X		
12. Software-engineering staff and managers conduct regular reviews to track technical progress, plans, performance, and issues against the development plan.				X							
13. Formal reviews, to address the accomplishments and results of project software engineering, are conducted at selected project milestones and at the beginning and completion of selected stages.			X								

Figure 2. A sample scoring worksheet. It can be used to summarize the score determined for the key activities of a given key process area, such as software project tracking and oversight, included in the SEI model. These scores are determined using the key-activity scoring guidelines shown in Figure 1.

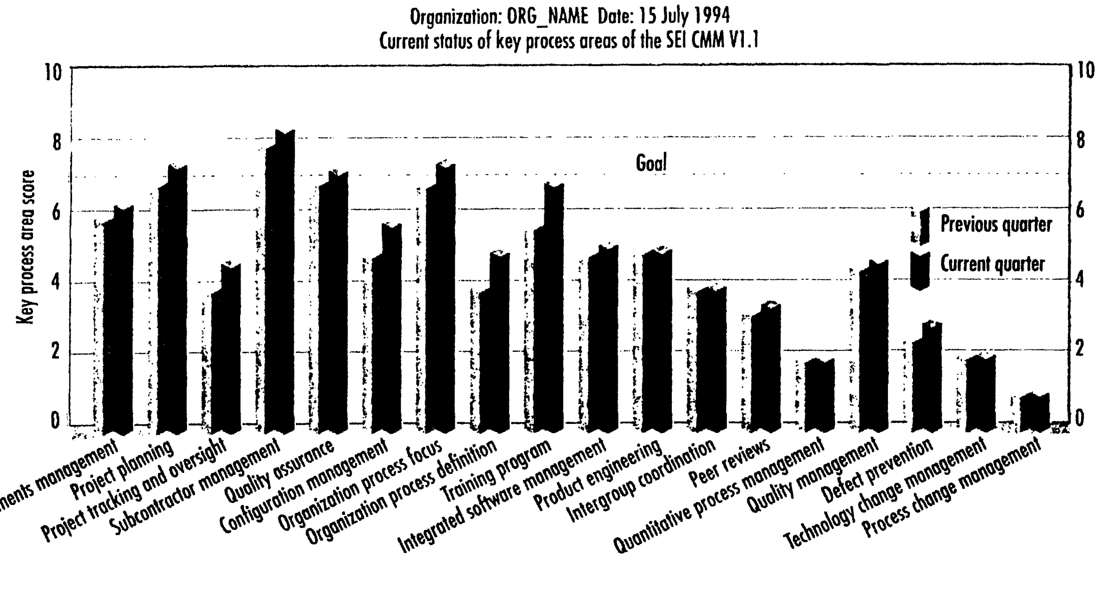


Figure 3. *Summarized progress report regarding SEI key process areas. Bars on the left correspond to the assessment results obtained the previous quarter. Bars on the right respond to the current quarter's results.*

a lengthy list of guidelines, such as one per key activity, which makes the scoring task more complex. The sample worksheet shown in Figure 2 is used to summarize the score obtained by using these guidelines. When applied at the project level, you translate the guideline "parts of the organization," as "subprojects" or "subsystems." When applied at the product-line, division, or group level, "parts of the organization" translates to "projects" or "project areas." Users of this scoring-guidelines matrix must ensure that they use the terms "commitment," "ability," "monitoring," and "verification" as described in the SEI model when determining a key activity's score.

To calculate the score for a specific key-process area, enter the score for each of its key activities in the worksheet shown in Figure 2. Average the individual key-activity scores to find the overall score for that key process area. Within Motorola, a score of 7 or higher for each key-process area at an SEI level i ($1< i \le 5$) indicates the organization will likely be assessed at SEI level i by a formal SEI assessment, assuming the organization has already been assessed as being at SEI level i–1. All the evaluation dimensions in Figure 1's scoring matrix carry equal weight when determining the score for a given key activity. Determine the key activity's score by examining all three evaluation dimensions and their scoring guidelines simultaneously.

Although each evaluation-dimension level represents a two-point increment, the score for a key activity can be an odd number if some of, but not all, the criteria for the next higher level are satisfied. For example, if some of the dimensions for a key process area are rated at the Fair level (4), while others are rated at the Marginally Qualified level (6), a score of 5 would be appropriate.

The average of the key process area scores for a given SEI level indicates how well the key process areas and activities corresponding to that level have been implemented within an organization. The key activities corresponding to each key process area in the CMM[3] are those listed in the sample worksheet. If multiple items are associated with an activity in the CMM, just consider them part of the package that describes the key activity when determining its score.

Low scores identify key activities and key process areas that need immediate attention to raise the organization's software-process capability. A low key-activity, key-process-area, or SEI-level score indicates a problem area that needs immediate attention and improvement. The next section provides an example of how the problem areas are highlighted within those Motorola business units that already use this method.

DATA PRESENTATION

The organization's current status, as determined using the scoring guidelines shown in Figure 1, are summarized using bar charts and/or Kiviat plots. The bar chart in Figure 3 summarizes the overall status of the key-process-area implementation. Note that a progress assessment and the presentation of the results may be done for a specific SEI level only, instead of all SEI CMM levels at the same time. Typically this is the next higher SEI level the organization is trying to achieve.

You can use Kiviat charts to summarize the status of a key-process-area implementation for a specific SEI level. Figure 4 is an example of an organization's progress in implementing CMM level 2. Each axis starting at the center of the circle corresponds to a key-process area at that level. This chart indicates the progress achieved during the chosen interval — in this case the last quarter — in advancing from level 1 to level 2. The chart also indicates the key process areas at level 2 for which additional focus is necessary, as well as those for which the improvement efforts have already paid off.

The same applies to higher CMM levels. Suppose management is not satisfied with the progress made on Software Project Tracking and Oversight and wants to obtain additional information about the key activities that must be immediately addressed.

Information on implementation status is presented in a bar chart like the one shown in Figure 5. The lower bars on this chart clearly indicate the key activities of the Software Project Tracking and Oversight key process areas that need immediate improvement. These activities include revisions to the development plan, senior management review of external commitments, communication of approved commitment changes, software-cost tracking, tracking software-engineering

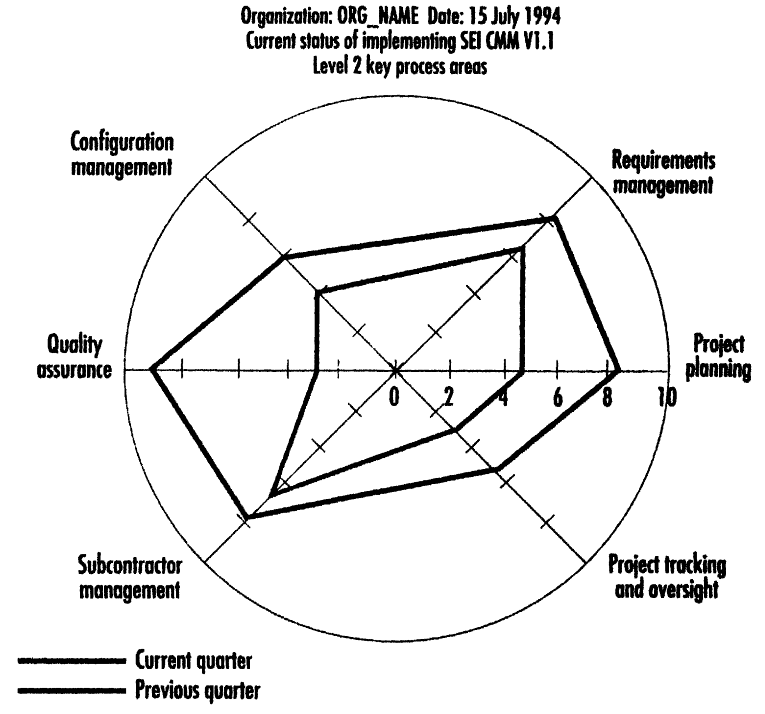


Figure 4. Summarized progress report for SEI level 2 key process areas.

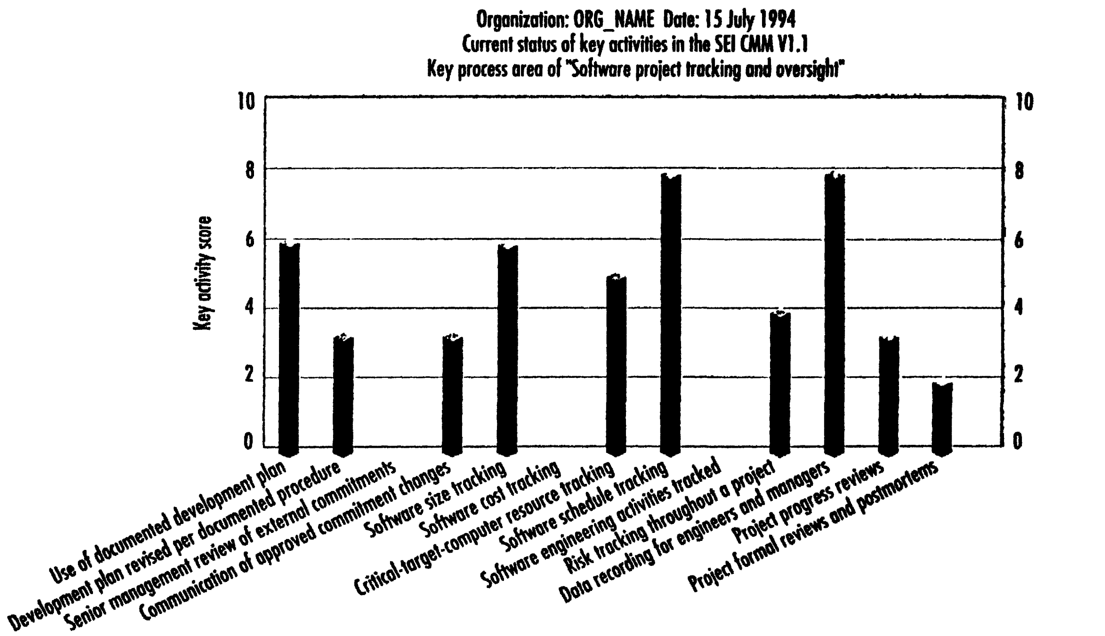
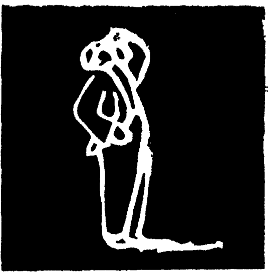


Figure 5. Status of key activities for the Software Project Tracking and Oversight key process area.



technical-activities, project-progress reviews, formal project reviews, and postmortems. Addressing these items will lead to better performance against the target of reaching higher SEI process-capability levels.

THIS PROCESS SPEEDS IMPROVEMENT BY PROVIDING A WAY TO MEASURE AND TRACK IT.

PARTICIPANTS

To be effective, the assessment instrument must be championed and used by members of the organization conducting a progress assessment:

♦ *Organization management.* This role is generally taken by senior management. They are primarily responsible for understanding what is involved in an SEI progress assessment, indicating their support for the whole process, committing resources to implement the action plan created, and following up to ensure completion.

♦ *Progress-assessment champion.* This role is critical. A single individual is responsible for championing the whole process (for the specific time period that it is done), ensuring organization management's support, identifying who within the organization should participate, taking care of administrative items, and championing the action-plan implementation. The champion should be in a technically competent middle-management position that is well-respected within the organization. This role requires a lot of work, and the champion can be involved in progress assessment only for his or her particular organization.

♦ *Progress-assessment facilitator.* This person is responsible primarily for ensuring that the progress assessment runs smoothly, providing consulting support when necessary. The progress assessment includes not just assessment meetings but also action-plan creation and implementation as a result of these meetings. The necessary background for the facilitator includes experience in conducting assessments and audits of software organizations. The facilitator may be involved in several progress assessments for different organizations at the same time. The progress-assessment facilitator and progress-assessment champion must cooperate closely. The facilitator must be more familiar with the instruments involved and the SEI model used, and have overall experience in evaluating organizations, such as using Motorola's Quality System Review[5] or other audit mechanisms.

♦ *Progress-assessment participants.* The assessment participants are primarily technical and middle-management software people involved in day-to-day software development and maintenance activities. They are not necessarily limited to software developers, testers, and managers; they can also be people working in product management, marketing, or other positions that are part of the overall organization. They participate in the entire progress assessment, including meetings and action-plan implementation.

♦ *Organization-improvement champion.* This person initiates the progress-assessment process during the preparation stage. If no organization-improvement champion exists, one must be identified who will initiate this process. Typically, he or she is also the progress-assessment champion, at least initially, when the progress assessment is introduced to an organization. Once the progress assessment is established, the progress-assessment champion may be changed every quarter to ensure wider participation.

PROCESS

The progress-assessment process provides an ordered series of activities that guide the participants in the use of the progress-assessment instrument. The process consists of four stages: preparation, assessment meeting, action plan and commitment, and follow-up.

Preparation. Activities at this stage focus on obtaining management buy-in, if it is not already obtained, and preparing to conduct an effective SEI progress assessment.

1. The organization-improvement champion meets with organization management to present the benefits of introducing SEI progress assessments and recommends their use.
2. The organization-improvement champion identifies a progress-assessment champion.
3. The progress-assessment champion identifies a progress-assessment facilitator.
4. The progress-assessment champion and the progress-assessment facilitator determine the scope of the SEI progress assessment.
5. The progress-assessment champion and organization management select the progress-assessment participants from the projects and groups included in the scope of the SEI progress assessment.

Assessment meeting. During this stage, participants agree on a scoring for the key process areas and activities and a list of strengths and weaknesses in these areas.

1. The progress-assessment champion conducts an overview session for the assessment meeting participants.
2. The assessment-meeting participants prepare for the assessment meeting, record their scores and findings in the worksheets, and forward them to the progress-assessment facilitator.
3. The progress-assessment facilitator uses a spreadsheet (or other tool) to summarize assessment-meeting-participant scores before the meeting.
4. The progress-assessment facilitator identifies a recorder for the assessment meeting.
5. The progress-assessment facilitator moderates the assessment meeting.

6. The recorder creates a draft list of scores and findings.

7. The progress-assessment champion moderates a review of this list.

8. The recorder updates and publishes the list.

9. The progress-assessment champion moderates a meeting with organization management and the progress-assessment participants where the scores and findings are presented.

Plan and commitment. At this stage the participants create the action plan, obtain commitments, and staff the plan according to the results of the assessment meeting.

1. The progress-assessment champion splits the progress-assessment participants into one team per key-process area. These teams generate draft action plans.

2. Each team meets with any existing organization key-process-area champions to ensure coordination and continuity of the action plans.

3. The draft action plans are reviewed and appropriately updated by the progress-assessment participants.

4. The progress-assessment champion ensures that the action plans are tracked using a project-management tool.

5. The progress-assessment champion moderates a meeting with organization management and progress-assessment participants, during which the action plans are presented and input is requested.

6. The action plans are updated on the basis of input by organization management.

7. Commitment templates for all action items are created and filled-in by the teams.

8. The progress-assessment champion ensures that individual meetings are scheduled with department managers to obtain their commitments.

9. Representatives of the teams participate in the meetings with department managers, finalize the commitment templates, and update their action plans appropriately.

Follow-up. During this final stage, participants ensure that the action plan is actually implemented and that sufficient progress is made, which is then reported to management.

1. Regular status meetings are conducted by each key-process-area team.

2. The progress-assessment champion conducts regular status meetings with the progress-assessment participants and provides status reports to organization management.

3. The organization-improvement champion identifies a new progress-assessment champion for next quarter's SEI progress assessment.

We have found that this process accelerates improvement by providing a way to measure it (the scoring guidelines) and track it (the presentation charts). This follows Motorola's approach to software measurement, which states: "Measurement is not the goal. The goal is improvement through measurement, analysis, and feedback."[6] The created action plans are shared with management, and requests for the necessary resources are made so that the actions can be implemented. This happens on a continuous basis, not just once every two years, which is the typical interval for formal SEI or other assessments. In fact, any actions necessary as a result of a formal SEI assessment or a Quality System Review may be folded into the already existing action plans developed through the use of SEI Progress Assessments.

MANAGEMENT BUY-IN IS ESSENTIAL FOR SUCCESSFUL PROGRESS ASSESSMENTS.

This process also provides a driver for continuous process improvement, in line with the spirit of Motorola's Quality System Review and other process- and quality-improvement initiatives.

LESSONS LEARNED

Management buy-in is essential to a successful implementation of the progress-assessment instrument and process. We introduced both at CIG's monthly Software Process Improvement meeting, explaining what the assessment instrument is and proposing its use to assess current status relative to SEI level 2 and to drive organizational improvement. Motorola CIG's management adopted this proposal and asked that each product group conduct their own self-assessment using this instrument, then create action plans for improvement. Regular action-plan status meetings were also requested and conducted by management to track improvement achieved over time. In the months that followed, we learned several important lessons about implementing this progress-assessment method.

♦ Determine before conducting a progress assessment what its scope is. Also, determine what management level will be considered as "senior" for progress-assessment purposes (director and above, for example). You need this information so that participants can obtain a common understanding of how the SEI CMM description applies to their organization. It also ensures consistency in the use of the scoring guidelines in Figure 1.

♦ Ensure that sufficient coverage is achieved across software-development and -maintenance functions and groups involved. Do this by carefully selecting the participants in the progress assessment of a given quarter. A group of five to six people should be sufficient. However, a larger group of about 20 may be used if you need to increase buy-in within the organization and ensure that the action-plan implementation will be staffed properly. A mix of experienced people who have participated in past SEI Progress Assessments and inexperienced people is recommended. In the case of a larger group, special attention is required by the progress-assessment facilitator to ensure that the meetings are sufficiently under control.

♦ To ensure proper coverage of the SEI CMM, use the following guideline: *All* SEI CMM sections for a given key process area, not just the "Activities Performed," should be considered when using the scoring guidelines to determine a score, and when the list of findings and the action plan are created. For example, items under "Ability to Perform" that are not evident in the organization should be listed in the list of findings and subsequently addressed through the action plan created.

♦ The progress-assessment facilitator should use the following method to reach consensus on the score for a key activity and speed up the meeting: Determine what the average suggested score by the participants is, then move higher or lower based on comments by the participants. Do this by first obtaining the individual participant scores prior to the progress-assessment meeting, then use a spreadsheet to determine the mean, standard deviation, and so forth, in advance.

♦ Ensure that the entire progress assessment focuses more on identifying the organization's strengths and weaknesses (the findings) and the implementation of the action plan created and less on what a given key activity's score should be.

In addition to Motorola's Cellular Infrastructure Group, several Motorola business units have adopted the use of SEI Progress Assessments, including product groups within the Satellite Communications Group, Semiconductor Products Sector, the Land Mobile Products Sector, and the Automotive and Industrial Electronics Group. Thus far, these groups' experiences with SEI Progress Assessments support the lessons learned within the CIG.

After using the progress-assessment process for several quarters, we were able to formally document it, which implies that it reflects a practically implemented sequence of steps rather than a list of steps that would be nice to do but have not been implemented yet.

Having already achieved SEI level 2 in the second quarter of 1993, work is already in progress for achieving SEI level 3 within the CIG, with the SEI Progress Assessment process continuing to be the key driver. Benefits similar to those reported by Raymond Dion[7] are anticipated as a result of achieving higher SEI process-maturity levels.

The instrument and process used for implementing the SEI Progress Assessment method can also be used in conjunction with additional models of software capability, quality, customer satisfaction, software measurements,[8] and so on, such as the Quality System Review to assess progress relative to "higher levels" in that model. I encourage you to use the SEI Progress Assessment method within your own organization and to share your results with other software practitioners in professional conferences and publications. ◆

ACKNOWLEDGMENTS

I thank Allan Willey and Kim Dobson for pointing out the need for a progress-assessment method and asking that one be developed. I would also like to thank Motorola management, especially CIG's, for supporting the use of progress assessments; Motorola's practitioners for taking ownership of the instrument and process, successfully completing several quarterly progress assessments, and creating and implementing action plans to drive the improvement of their own product group/organization over time; and Bob Yacobellis, senior member of the technical staff and manager, Corporate Software Process Engineering Group, for his constructive review and comments regarding earlier versions of the SEI Progress Assessment method.

REFERENCES

1. M.C. Paulk et al., "Capability Maturity Model for Software," Tech. report CMU/SEI-91-TR-24, Software Eng. Inst., Pittsburgh, 1991.
2. M.C. Paulk et al.,"Capability Maturity Model for Software, Version 1.1,"Tech. report CMU/SEI-93-TR-24, Software Eng. Inst., Pittsburgh, 1993.
3. M.C. Paulk et al., "Capability Maturity Model, Version 1.1," *IEEE Software*, July 1993, pp. 18-27.
4. A. Topper and P. Forgensen, "More than One Way to Measure Process Maturity," *IEEE Software*, Nov. 1991, pp. 9-10.
5. Motorola Corporate Quality Council, "Motorola Corporate Quality System Review Guidelines," Revision 1, Literature # BR1202/D, Phoenix, Ariz., 1991.
6. M. K. Daskalantonakis, "A Practical View of Software Measurement and Implementation Experiences Within Motorola," *IEEE Trans. Software Eng.*, Nov. 1992, pp. 998-1010.
7. R. Dion, "Process Improvement and the Corporate Balance Sheet," *IEEE Software*, July 1993, pp. 28-35.
8. M.K. Daskalantonakis, V.R. Basili, and R.H. Yacobellis, "A Method for Assessing Software Measurement Technology," *Quality Engineering Journal*, Vol. 3, No. 1, 1990, pp. 27-40.

Address questions about this article to Daskalantonakis at Cellular Infrastructure Group, Motorola, Inc., 1501 West Shure Dr., Arlington Heights, IL 60004; dask@mot.com.

Successfully Applying Software Metrics

Robert B. Grady, Hewlett-Packard

What do you need to measure and analyze to make your project a success? These examples from many projects and HP divisions may help you chart your course.

The word *success* is very powerful. It creates strong, but widely varied, images that may range from the final seconds of an athletic contest to a graduation ceremony to the loss of 10 pounds. Success makes us feel good; it's cause for celebration.

All these examples of success are marked by a measurable end point, whether externally or self-created. Most of us who create software approach projects with some similar idea of success. Our feelings from project start to end are often strongly influenced by whether we spent any early time describing this success and how we might measure progress.

Software metrics measure specific attributes of a software product or a software-development process. In other words, they are measures of success. It's convenient to group the ways that we apply metrics to measure success into the four areas shown in Figure 1. This article contains four major sections highlighting examples of these areas.

Figure 1 also shows two arrows that represent conflicting pressures for data. For example, on one of the first software projects I managed, the finance department wanted me to use their system to track project costs, arguing that this would help me. I shortly learned that their system didn't give me the kind of information I needed to be successful. The reports weren't timely or accurate, and they didn't usefully measure progress. This was one of my first experiences with the opposing desires for information that can arise between a project manager and the division's management team. They wanted summary data across many diverse functions: I wanted data that would help me track day-to-day progress.

I soon realized that projects stand the best chance of success when the goals driving the use of different measures can be stated and mutually pursued. This article's examples are all from real projects, and they were chosen to show both viewpoints illustrated by the arrows in Figure 1. The examples also show how the possibly con-

Reprinted from *Computer*, Vol. 27, No. 9, Sept. 1994, pp. 18–25.

flicting goals of a project team and of an organization can effectively complement each other. Finally, they are examples of things that *you* can measure to be more successful.

Project estimation and progress monitoring

Today there are dozens of software-estimation tools. Figure 1 suggests that such tools can be quite useful to project managers. These tools are now very sophisticated because they account for many possible project variables. Unfortunately, most of us are not much better at guessing the right values for these variables than we are at guessing total project schedules.

The basis for estimates. Most estimating tools are based on limited measurements. For example, the first three columns of Table 1 show measurements for my early 25,000-engineering-hour project, with and without nonengineering activities. (I finally tracked these measurements without using our normal accounting system.) Some of the data is useful for future estimates. For example, the percentages for supervision and administrative support would be reasonably accurate for other projects, particularly since they can be controlled. Even the time spent in different activities doesn't differ much from the averages for 132 more-recent Hewlett-Packard projects, although my team didn't collect the data in exactly the way that HP currently does.

Should you collect data like this for your projects? Since estimation models are based on such data, informally collecting it will help you track the validity of your inputs into any model. This data can give you useful insights into the accuracy of your estimates. The earlier you find differences, the more likely it is that management might accept schedule changes.

The bottom line. Higher level managers are usually not interested in as much detail as Table 1 presents. They want the bottom line: Is the project on schedule? Figure 2 shows how one HP lab tracked this across many projects.[1,2] Two ideas went into this graph. First, a schedule slip is the amount of time that a project schedule is moved to a later date. Second, average project progress for a

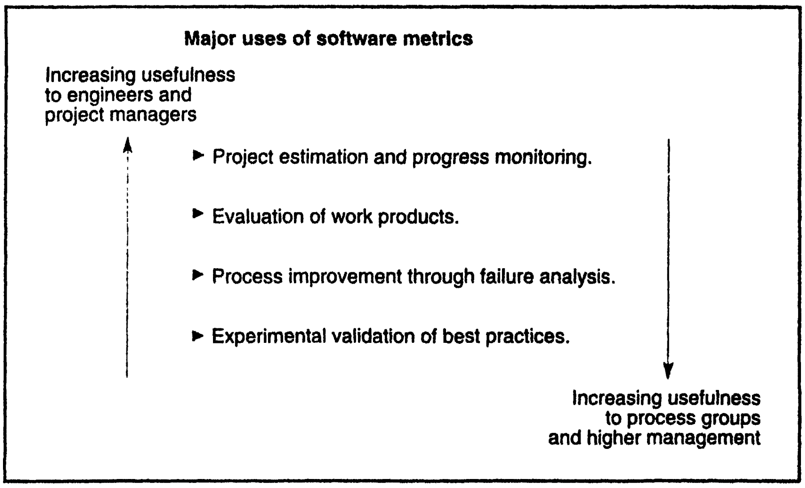

Major uses of software metrics
Increasing usefulness to engineers and project managers
► Project estimation and progress monitoring.
► Evaluation of work products.
► Process improvement through failure analysis.
► Experimental validation of best practices.
Increasing usefulness to process groups and higher management

Figure 1. Major uses of software metrics and the conflicting pressures for data.

Table 1. Task breakdowns for a 25,000-engineering-hour software project over 2.5 calendar-years.

Tracked Times	Project %	Eng. Only %	Categories Currently Tracked	Approx. Project %	HP Average
Investigation	20	26	Reqs./Specs.	19	18
External/Internal Reference Specs	2	2	Design	16	19
Coding	19	23	Implement	32	34
Debugging	19	24			
Integration	11	14	Test	33	29
Quality Assurance	8	11			
Manuals	7				
Supervision	9			Not included	
Support	5				

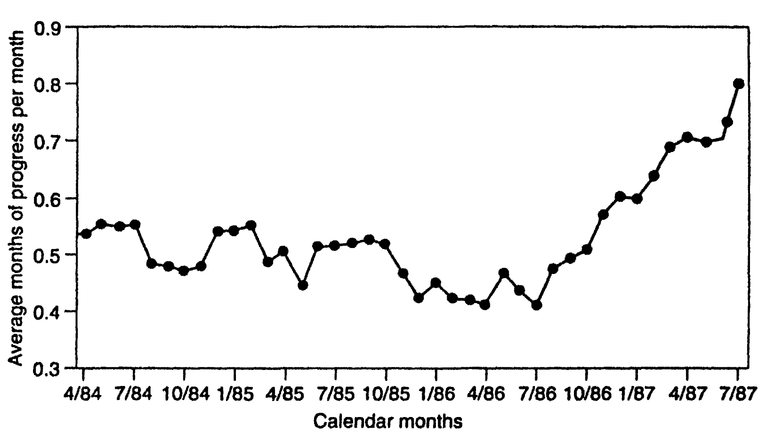


Figure 2. Development project progress for all software projects in one HP division.

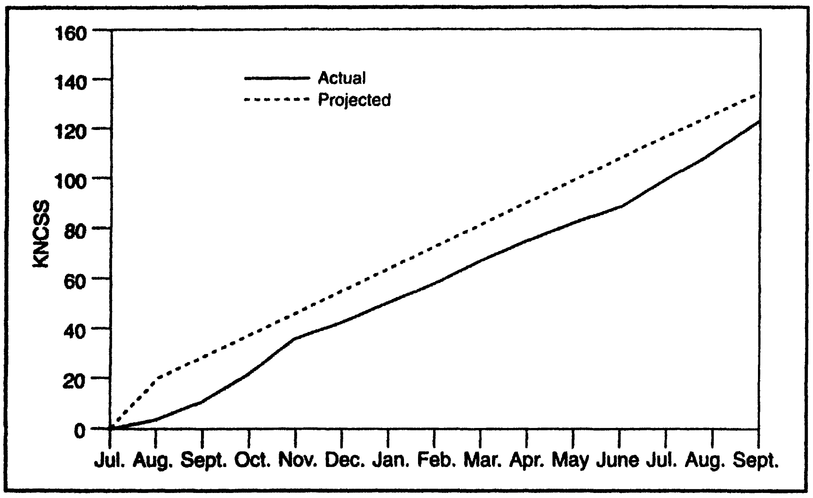


Figure 3. Plot of thousands of noncomment source statements (KNCSS) against time for project summarized in Table 1 (© 1987 Prentice-Hall, used with permission).

time period is defined as one minus the ratio of the sum of all project slips divided by the sum of project elapsed times.

For example, suppose you have a small lab with three projects. In a one-month period, the first project's manager believes that it is on schedule (its schedule doesn't change). Its slip is zero. The second project had a bad month. Its slip is one. The third project's manager expects the project will slip one week, for a slip of about one-quarter. The sum of the slips is 1.25. The sum of elapsed times is 3. Average project progress is therefore 1 - (1.25/3) = 0.58.

The graph uses a moving average to smooth month-to-month swings. The lab first started plotting the graph around October 1985. They had enough historical data to show that they had only averaged about a half month of progress for every elapsed month. After monitoring the graph for a few months, the project teams gradually focused on more accurate schedules, and they improved their accuracy to an enviable point.[1,2]

Successful usage. The examples shown in Table 1 and Figure 2 were both successes. They show two things that you will see repeated in other examples:

- Lab management wanted limited, high-level summary data.
- Project-management data provided both confidence-building tracking information and a basis for better future estimates.

A major reason for success in both cases was that their end points — their goals — were measurable. Figure 2 graphically shows that the way the division estimated schedules hadn't changed for at least 1.5 years before they defined a way to measure progress. In the other example, Table 1, the data influenced dozens of my decisions. They included resource balancing, intermediate and final schedule commitments, test schedules, and technical writing schedules. Furthermore, I could confidently show and explain progress on a very large project in ways that few high-level HP managers had seen before.

Monitoring progress against estimates. There are two time-proven ways to track progress on a software project. The first is to track completed functionality (the features or aspects of the software product).

Tracking functionality. Despite often-expressed concerns about the usefulness of counting lines of code, I have found tracking code size against time to be very useful for managing projects. Figure 3 is a plot of thousands of noncomment source statements (KNCSS) against time for the project summarized in Table 1.[3]

KNCSS represents completed functionality here. It is reasonable to use during the coding and testing phases, particularly if you track coded NCSS separately from tested NCSS (not shown in Figure 3). I updated this graph every week to make sure the project was on track. I also tracked the status of modules designed for this project. This provided a link to our original estimates by exposing design areas significantly different from earlier plans.

More recently, HP has measured FURPS criteria (functionality, usability, reliability, performance, and supportability) to complement simpler size-tracking metrics. (Grady[2] describes FURPS more completely.)

Finally, function points, another popular functionality measure, are computed from a combination of inputs, outputs, file communications, and other factors. They can be computed independently of source code, so some people find their added difficulty of use offset by this earlier availability. (Capers Jones' *Applied Software Measurement,* McGraw-Hill, New York, 1991, is a useful function-point reference.)

Found-and-fixed defects. Tracking functionality doesn't attract high-level management attention like the trends of found-and-fixed defects. This second method is very useful in monitoring progress in later development phases. These trends are also among the most important aids you have in deciding when to release a product successfully. Methods for analyzing such trends vary. Variations include simple trend plotting, sophisticated customer-environment modeling, and accurate recording of testing or test-creation times.

Using defect trends to make larger system-release decisions gives valuable confidence to higher level managers. They feel more comfortable when major release decisions are backed by data and graphs. Figure 4 shows the system test defects for a project involving over 30 engineers. It shows this project's status about one month before completion. The team derived the goal from past project experiences. One release criteria was for the defects/1,000-test-hours rate to drop and stay below the goal line for at least two weeks before release. The alternative projections were simple hand-drawn extrapolations using several of the past weeks' slopes. The team updated the graph every week. The weekly test hours were much fewer than 1,000, so the weekly ratios may give an impression that there were more defects than there really were.

HP has learned that the critical downward trend that Figure 4 displays is necessary to avoid costly postrelease crises. This project's downward slope continued, and the project released successfully. An opposite example was an HP software system released despite an absence of a clear downward defect trend. The result was a multimillion-dollar update shortly after release and a product with a bad quality reputation. This kind of mistake can cause

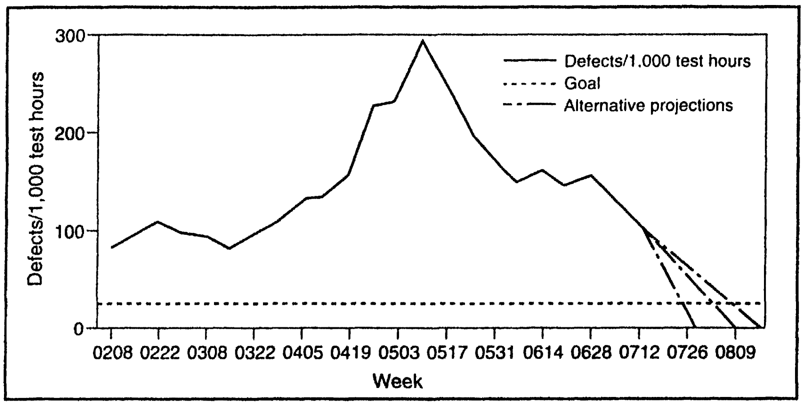


Figure 4. System-test defect trend for project involving over 30 engineers.

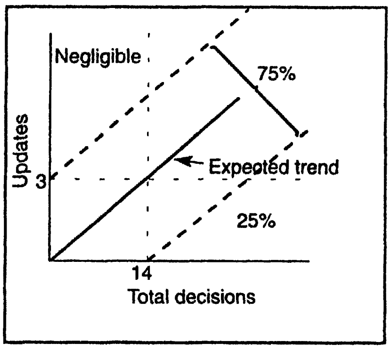


Figure 5. Trend analysis of number of source updates for system-level testing versus the number of decision statements per subroutine.

an entire product line's downfall. A recent article describes how one company learned this lesson the hard way.[4]

Plotting defect trends is one method where both project and high-level management have similar interests. While exact completion points may vary based on differing project goals, better decisions are possible when trends are visible. Those decisions will help to ensure project success.

Evaluation of work products

A work product is an intermediate or final output that describes the design, operation, manufacture, or test of some portion of a deliverable or salable product. It is not the final product. All software development finally results in a work product of code. While this section's brief examples center on code, the idea of extracting useful metrics from virtually any work product is the point to remember.

Because code can be analyzed automatically, it has been a convenient research vehicle for sophisticated statistical analysis. Unfortunately, this emphasis has created a strong bias in perceptions of metric applications. Many managers believe that useful metrics require time-consuming techniques outside of their normal decision-making processes. Even recently, one metrics expert told me that the minimum number of code metrics a project manager should monitor is around 20.

Cyclomatic complexity. Fortunately, HP has had good results when measuring just one code metric: cyclomatic complexity, which is based on a program's decision count. (The decision count includes all programmatic conditional statements, so if a high-level-language statement contains multiple conditions, each condition is counted once.) One HP division especially saw this when they combined the metric with a visual image that graphically showed large complexity.[5] Graphs, and their source code cross references, help engineers understand problem locations and may provide insights for fixing them. The graphs excite managers because their availability encourages engineers to produce more maintainable software. This doesn't mean that the tool's numeric values are ignored: Complexity metrics give managers and engineers simple numerical figures of merit.

When considering engineering tool value, high-level managers want to know whether using such a tool yields better end products in less time. Project managers may have to look at other data like that shown in Figure 5 to build a strong case for tools.[6] This study concerned a project of 830,000 lines of executable Fortran code.

Those doing the study plotted the relationship between program-decision counts and the number of updates reflected by their source code control system. Seventy-five percent of the updates fell within the dashed lines. For their system, the number of updates was proportional to the number of decision statements. From their analysis, they drew a trend line. By knowing the cost and schedule effects of modules with more than three updates, they concluded that 14 was the maximum decision count to allow in a program. (Tom McCabe originally suggested 10, based on testing difficulty.[7])

You can do a similar analysis or you can accept these and similarly documented results and assume they apply equally well to your project. Then estimate how using complexity tools can make your project more effective. Measure normal defect rates and both the engineering time and the calendar time to do fixes. Estimate how long it would take to run complexity tools. Finally, calculate your savings when you reduce complexity *before* your people start finding the defects in test. Grady[2] provides an economic justification for the purchase of complexity tools like these.

But the metrics expert who set a minimum of 20 code metrics was not totally wrong. Because cyclomatic complexity is a measure of control complexity, it is more valuable for control-oriented applications than for data-oriented ones. It works for both, but the characteristics of data-oriented applications suggest that you must consider other dimensions as well. Unfortunately, reported data-oriented results haven't been as thoroughly tested as those I've mentioned.

Design complexity. A promising metric for data-oriented complexity is fanout squared. The fanout of a module is the number of calls from that module. At least three studies have concluded that fanout squared is one component of a design metric that correlates well to probability of defects.[2,8,9] More importantly, fanout squared can be determined before code is created. Figure 6 on the next page shows the top-level structure chart of the most defect-prone module in a system. This module was the source of 50 percent of the system test defects, even though it had only 8 percent of the code. Its fanout squared was also the largest among the system's 13 top-level modules. In fact, postrelease defect densities were highly correlated to the fanout squared of the system's modules.

The figure shows a large number of

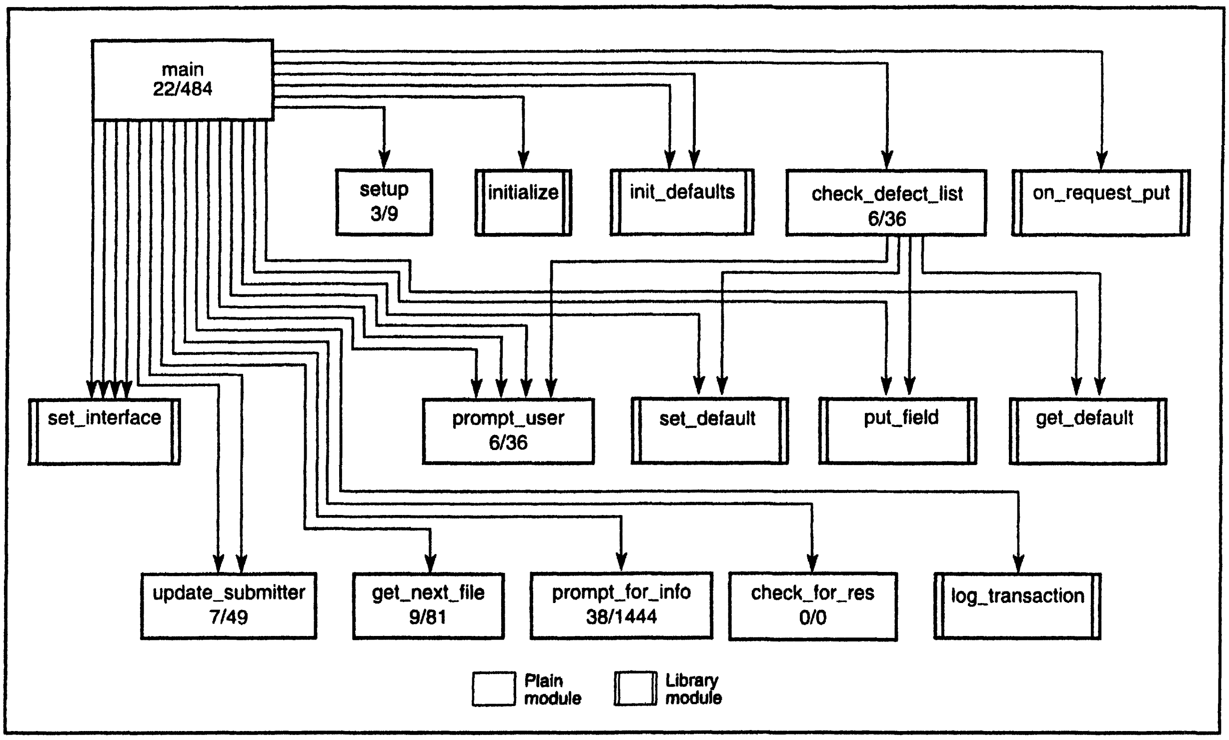


Figure 6. Structure chart showing fanout/fanout squared for each module. Diagram is simplified to show a complete set of connections for the main module only. Calls to system or standard utility routines are not shown or counted.

connections between *main* and the 15 other modules. The library modules don't call other modules, so no fanout is given for them. Note the fanout for *prompt_for_info*. With a fanout of 38 (and a fanout squared of 1,444), you can imagine its structure chart's complexity. Such large fanouts suggest that there is a missing design layer. Structure charts combine with fanout squared to give the same type of results as cyclomatic complexity and graphical views of control flow, only earlier.

Based on limited past experimental results, design complexity metrics may not be justified yet during design. However, if information like fanout squared were readily available as a byproduct of normal project tools, progress toward understanding design complexity would be faster. Meanwhile, measuring code complexity is desirable from both project and higher management viewpoints. Also, measuring design complexity in designs provides an important research opportunity.

More on successful usage. Cyclomatic complexity and fanout squared are just two types of work-product analysis. Automation recently introduced by the CASE (computer-aided software engineering) boom continues to expand engineers' comprehension of their work. You can take advantage of complexity data to help your project in several ways. Like the Figure 5 example suggests, you can enforce a standard by computing complexity for all modules and not accepting any above some value. Another approach is to limit the number of modules above a given complexity level. Then make sure those modules are inspected and tested carefully. Another way is to require more documentation for such modules. Whichever way works best for you, it is certain that complexity information will help both you and your engineers to be more successful by providing information for better, more timely decisions.

Process improvement through failure analysis

I believe this third area from Figure 1 is the most promising for improving development processes. Failure analysis, and finding and removing major defect sources, offers the best short-term potential for guiding improvements.

Project defect patterns. There are several valuable approaches. One simple approach for project managers is to monitor the number of defects found during system test, code inspections, or design inspections. This data can be sorted by module, and special actions can be taken as soon as potential problem areas appear. For example, Figure 7 shows both prerelease system-test defects and defects for a product during the first six months after its release.[3]

Unfortunately, the project manager in this case didn't do anything different until after collection of postrelease data. The project manager then focused team effort on thorough inspections and better testing of the three most defect-prone modules. (While these modules were only 24 percent of the code, they accounted for 76 percent of the defects.) As a result, the team succeeded in greatly reducing the product's incoming defect rate by focusing on those three modules.

Software process defect patterns. Another type of failure analysis examines defect patterns related to development processes. This analysis affects enough people to generally require lab-level management sponsorship. Many HP divisions today start this analysis by cate-

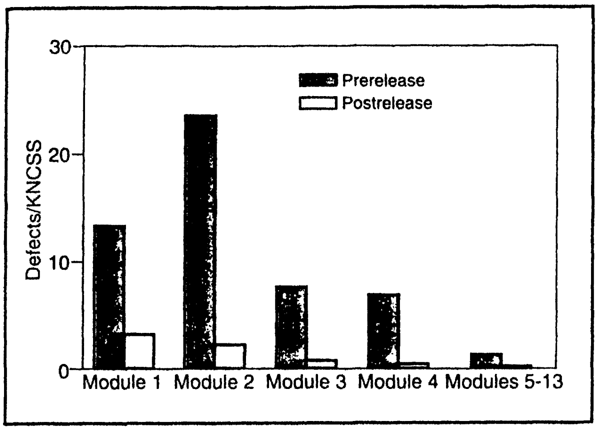


Figure 7. Analysis by code module for prerelease system-test defects and for those occurring during the first six months after the product's release (© 1987, Prentice-Hall, used with permission).

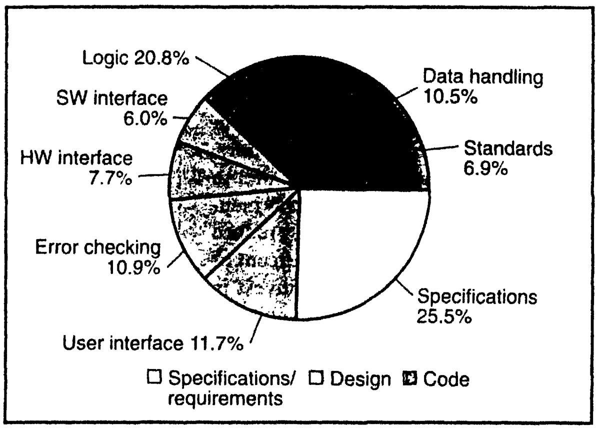


Figure 8. Top eight causes of defects for four Scientific Instruments Division projects at Hewlett-Packard.

gorizing their defects according to a three-level HP model used since 1987. The three levels are origin, type, and mode. Grady[2] describes several significant improvements achieved in divisions using this model. Figure 8 shows recent data for two levels of the model for yet another HP division (Scientific Instruments Division). The shading represents defect origin information, and the pie wedges are defect types. The data reflects the eight most frequently recorded causes of defects for four projects.

The division performed three post-project reviews to brainstorm potential solutions to their top four defect types. Several initiatives were launched. The first of these that yielded data for a full project life cycle recently concluded. The project team had decided to focus on user-interface defects. They had over 20 percent of that defect type on their previous project (even though the division-wide average was lower). They brainstormed the Figure 9 fishbone diagram and decided to create guidelines for user-interface designs that addressed many of the fishbone-diagram branches.

Their results were impressive. They reduced the percentage of user-interface defects in test for their new year-long project to roughly 5 percent. Even though the project produced 34 percent *more* code, they spent 27 percent *less* time in test. Of course, other improvement efforts also contributed to their success. But the clear user-interface defect reduction showed them that their new guidelines and the attention they paid to

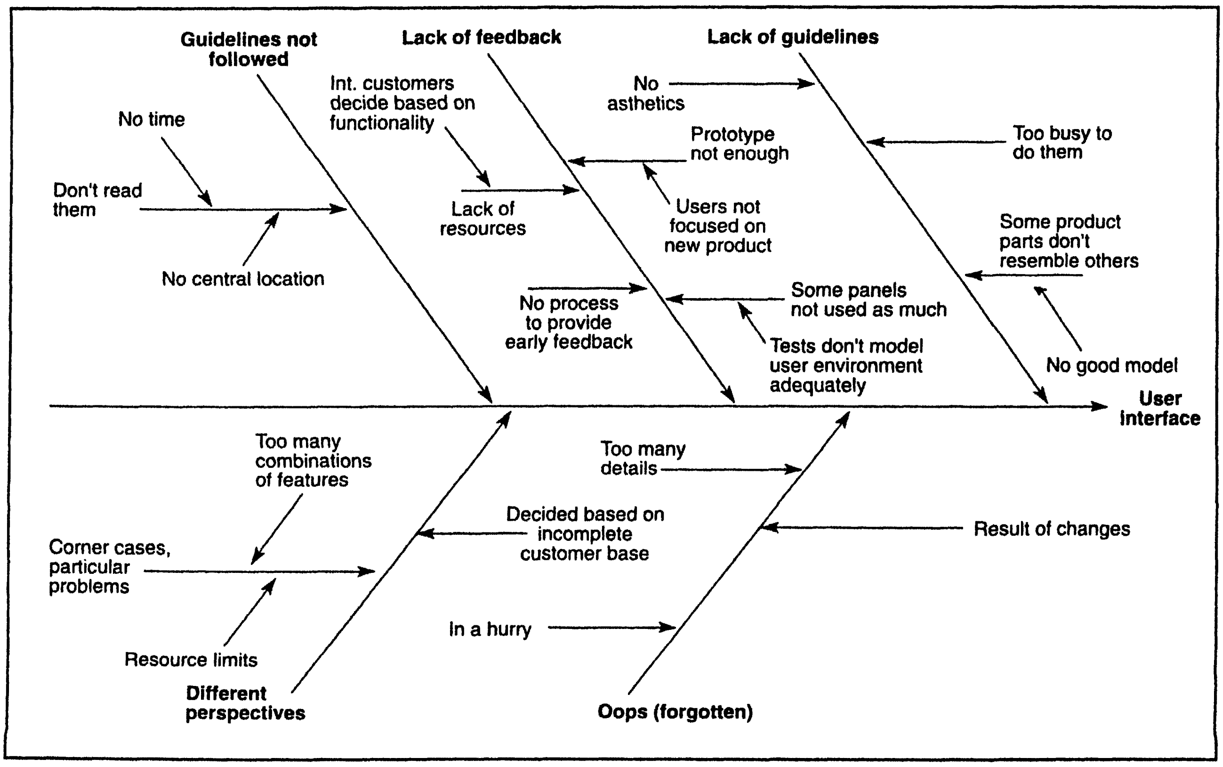

Guidelines not followed
No time
Don't read them
No central location
Lack of feedback
Int. customers decide based on functionality
Lack of resources
No process to provide early feedback
Lack of guidelines
No asthetics
Prototype not enough
Users not focused on new product
Some panels not used as much
Tests don't model user environment adequately
Too busy to do them
Some product parts don't resemble others
No good model
User Interface
Too many combinations of features
Corner cases, particular problems
Resource limits
Different perspectives
Too many details
Decided based on incomplete customer base
In a hurry
Result of changes
Oops (forgotten)

Figure 9. Fishbone diagram showing the causes of user-interface defects.

their interfaces were major contributors.

The examples you've seen in Figures 7, 8, and 9 show how a small investment in failure analysis can reap practical short-term gains. Ironically, the main limiter to failure-analysis success is that many managers still believe they can quickly reduce *total* effort or schedules by 50 percent or more. As a result, they won't invest in more modest process improvements. This prevents them from gaining 50 percent improvements through a series of smaller gains. Because it takes time to get any improvement adopted *organization-wide*, these managers will continue to be disappointed.

Experimental validation of best practices

This software metric use has been the most successful of the four listed in Figure 1. People have validated the success of important engineering practices (for example, prototyping,[10] reducing coupling, increasing cohesion,[9] limiting complexity,[6] inspections and testing techniques,[11] and reliability models[12]). This validation should lead to quicker, widespread acceptance of these "best" practices.

Of the four Figure 1 metric uses, project managers are least motivated to validate best practices because normal project demands have higher priority. On the other hand, these metrics have probably brought project managers the greatest benefits. The first example here is what high-level managers want to see. One HP division measured the data in Table 2 for different test and inspection techniques. The average efficiency of code reading/code inspections was 4.4 times better than other test techniques yield.[2,13] This data helps project managers to plan inspections for their projects and to convince their engineers of the merits of inspections by showing the benefits.

Table 2. Comparison of testing efficiencies.*

Testing Type	Efficiency (Defects found/hour)
Regular use	0.210
Black box	0.282
White box	0.322
Reading/ Inspections	1.057

*Defect-tracking system lumped code reading and inspections into one category. About 80 percent of the defects so logged were from inspections.

However, experience has shown that it takes many years to widely apply even proven best practices. It often takes local proof to convince engineers to change their practices. For example, Henry and Kafura first showed the fanout squared metric discussed earlier to be useful over 10 years ago (as a part of their information-flow metric).[8] Even then, they pointed out how such an early design metric would be useful during design inspections. Unfortunately, most software-developing organizations don't have standard design practices yet. Also, people haven't been convinced that it's worth the effort to do high-level design with the detail necessary to compute such measures.

Project managers might find Figure 10's graph useful for their projects. It shows all the fanout squared values for an HP product. If you had this information early in your project, you could focus inspections and evaluations on the high-value modules. Like cyclomatic complexity, fanout squared appears to have several very desirable properties:

- It is easy to compute.
- Graphical views (like Figure 6) do reflect high complexity.
- The metric exposes a small percentage of a system's modules as potential problems.

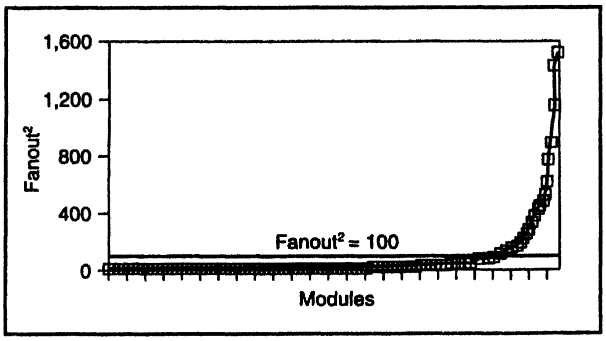


Figure 10. Fanout squared for a 250-module product, sorted by ascending fanout squared (which is more than 100 for only 10 percent of the modules).

Although this may be a significant future metric, it illustrates a dilemma. How much time can project managers or organizations spend proving such practices? As positive evidence grows and competitive pressures for higher quality grow, the motivation to apply promising new practices also increases. Not all validations of beneficial practices are as easy to measure as inspections. However, this is the road to progress. My advice to project managers is to invest some of your team's effort on improvements, but track and validate the benefits. My advice to high-level managers is to reserve some funds and encouragement to support such validations.

How do you apply software metrics to be successful? Review the four major uses of metrics, studying the project-level and management-level examples from successful projects. These examples lead to three recommendations for project managers:

- Define your measures of success early in your project and track your progress toward them.
- Use defect data trends to help you decide when to release a product.
- Measure complexity to help you optimize design decisions and create a more maintainable product.

Don't forget that other aspects contribute to successful metrics usage and project management beyond this article's examples. They include linking metrics to project goals, measuring product-related metrics, and ensuring reasonable collection and interpretation of data.

Consider two more recommendations for strategic purposes:

- Categorize defects to identify product and process weaknesses. Use this data to focus process-improvement decisions on high-return fixes.
- Collect data that quantifies the success of best practices.

This is all useful advice, *but what do you need to measure to be successful?* It is difficult to reduce this answer to a small set of measures for high-level managers. Chapter 15 of Grady[2] discusses nine useful management graphs. Finally, I suggest that project managers collect the following data:

- engineering effort by activity,
- size data (for example, noncomment source statements or function points),
- defects counted and classified in multiple ways,
- relevant product metrics (for example, selected measurable FURPS),
- complexity, and
- testing code coverage (an automated way of measuring which code has been tested).[2]

Understand how each of these relates to *your* success, and perform timely analyses to optimize your future. ■

Acknowledgments

I especially thank Jan Grady and Debbie Caswell for their early reviews and suggestions when this article was just a way of organizing my ideas for a talk. I also thank other Hewlett-Packard and *Computer* reviewers for their helpful comments and suggestions. Finally, thanks to Brad Yackle, Mark Tischler, and the others at SID for sharing their failure-analysis results.

References

1. D. Levitt, "Process Measures to Improve R&D Scheduling Accuracy," *Hewlett-Packard J.*, Vol. 39, No. 2, Apr. 1988, pp. 61-65.

2. R. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Englewood Cliffs, N.J., 1992.

3. R. Grady and D. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice-Hall, Englewood Cliffs, N.J., 1987, pp. 31 and 110.

4. D. Clark, "Change of Heart at Oracle Corp.," *San Francisco Chronicle*, July 2, 1992, pp. B1 and B4.

5. W. Ward, "Software Defect Prevention Using McCabe's Complexity Metric," *Hewlett-Packard J.*, Vol. 40, No. 2, Apr. 1989, pp. 64-69.

6. R. Rambo, P. Buckley, and E. Branyan, "Establishment and Validation of Software Metric Factors," *Proc. Int'l Soc. Parametric Analysts Seventh Conf.*, 1985, pp. 406-417.

7. T. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, Vol. SE-2, No. 4, Dec. 1976, pp. 308-320.

8. S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Trans. Software Eng.*, Vol. SE-7, No. 5, Sept. 1981, pp. 510-518.

9. D. Card with R. Glass, *Measuring Software Design Quality*, Prentice-Hall, Englewood Cliffs, N.J., 1990.

10. B.W. Boehm, T. Gray, and T. Seewaldt, "Prototyping vs. Specifying: A Multi-Project Experiment," *Proc. Seventh Int'l Conf. Software Eng.*, IEEE Press, Piscataway, N.J., Order No. M528 (microfiche), 1984, pp. 473-484.

11. L. Lauterbach and W. Randell, "Six Test Techniques Compared: The Test Process and Product," *Proc. Fourth Int'l Conf. Computer Assurances*, Nat'l Inst. Standards and Technology, Gaithersburg, Md., 1989.

12. M. Ohba. "Software Quality = Test Accuracy × Text Coverage," *Proc. Sixth Int'l Conf. Software Eng.*, IEEE Press, Piscataway, N.J., 1982, pp. 287-293.

13. T. Tillson and J. Walicki, "Testing HP SoftBench: A Distributed CASE Environment: Lessons Learned," *HP SEPC Proc.*, Aug. 1990, pp. 441-460 (internal use only).

Using Metrics to Manage Software Projects

Edward F. Weller

Bull HN Information Systems*

In 1989, Bull's Arizona facility launched a project management program that required additional software metrics and inspections. Today, the company enjoys improvements in quality, productivity, and cost.

Five years ago, Bull's Enterprise Servers Operation in Phoenix, Arizona, used a software process that, although understandable, was unpredictable in terms of product quality and delivery schedule. The process generated products with unsatisfactory quality levels and required significant extra effort to avoid major schedule slips.

All but the smallest software projects require metrics for effective project management. Hence, as part of a program designed to improve the quality, productivity, and predictability of software development projects, the Phoenix operation launched a series of improvements in 1989. One improvement based software project management on additional software measures. Another introduced an inspection program,[1] since inspection data was essential to project management improvements. Project sizes varied from several thousand lines of code (KLOC) to more than 300 KLOC.

The improvement projects enhanced quality and productivity. In essence, Bull now has a process that is repeatable and manageable, and that delivers higher quality products at lower cost. In this article, I describe the metrics we selected and implemented, illustrating with examples drawn from several development projects.

Project management levels

There are three levels of project management capability based on software-metrics visibility. (These three levels shouldn't be equated with the five levels in the Software Engineering Institute's Capability Maturity Model.) Describing them will put the Bull examples in perspective and show how we enhanced our process through gathering, analyzing, and using data to manage current projects and plan future ones.

First level. In the simplest terms, software development can be modeled as shown in Figure 1. Effort, in terms of people and computer resources, is put into a process that yields a product. All too often, unfortunately, the process can only be described

* Since writing this article, the author has joined Motorola.

Reprinted from *Computer*, Vol. 27, No. 9, Sept. 1994, pp. 27–33.

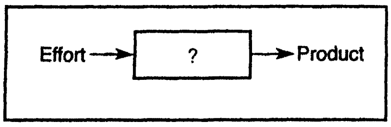


Figure 1. Software development level 1: no control of the development process. Some amount of effort goes into the process, and a product of indeterminant size and quality is developed early or (usually) late, compared to the plan.

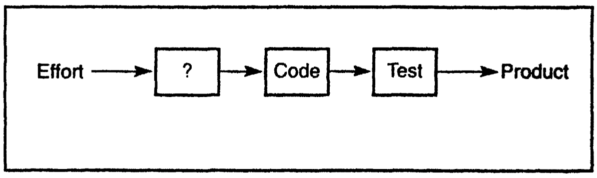


Figure 3. Software development at level 2: measurement of the code and test phases begins.

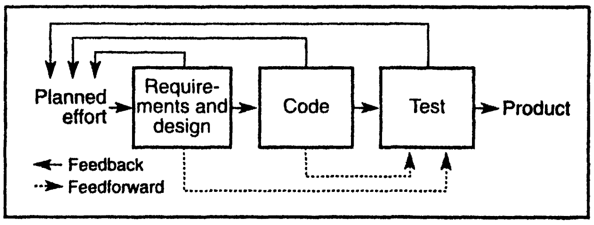


Figure 4. Software development level 3: control of the entire development process. You measure the requirements and design process to provide feedforward to the rest of the development as well as feedback to future planning activities.

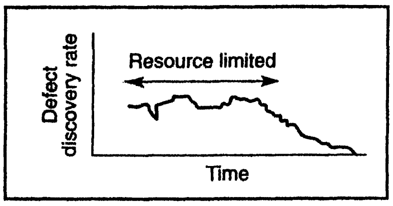


Figure 2. Defect discovery profile for lower development levels. The number of defects in the product exceeds the ability of limited resources to discover and fix defects. Once the defect number has been reduced sufficiently, the discovery rate declines toward zero. Predicting when the knee will occur is the challenge.

by the question mark in Figure 1. Project managers and development staff do not plan the activities or collect the metrics that would allow them to control their project.

Second level. The process depicted in Figure 1 rarely works for an organization developing operating-system software or large applications. There is usually some form of control in the process. We collected test defect data in integration and system test for many years for several large system releases, and we developed profiles for defect removal that allowed us to predict the number of weeks remaining before test-cycle completion. The profile was typically flat for many weeks (or months, for larger system releases in the 100- to 300-KLOC range) until we reached a "knee" in the profile where the defect discovery rate dropped toward zero (see Figure 2).

Several factors limit the defect discovery rate:

- Defects have a higher probability of being "blocking defects," which prevent other test execution early in the integration- and system-test cycle.
- The defect discovery rate exceeds the development staff's capacity to analyze and fix problems, as test progresses and more test scenarios can be run in parallel.

Although this process gave us a fairly predictable delivery date once the knee was reached, we could not predict when the knee would occur. There were still too many variables (as represented by the first two boxes in Figure 3). There was no instrumentation on the requirements or design stages (the "?" in Figure 3).

Our attempts to count or measure the size of the coding effort were, in a sense, counterproductive. The focus of the development effort was on coding because code completed and into test was a countable, measurable element. This led to a syndrome we called WISCY for "Why isn't Sam coding yet?" We didn't know how to measure requirements analysis or design output, other than by document size.

We also didn't know how to estimate the number of defects entering into test. Hence, there was no way to tell how many weeks we would spend on the flat part of the defect-removal profile. Predicting delivery dates for large product releases with 200 to 300 KLOC of new and changed source code was difficult, at best.

A list of measures is available in the second-level model (Figure 3):

- effort in person-months,
- computer resources used,
- the product size when shipped, and
- the number of defects found in the integration and system tests.

Although these measures are "available," we found them difficult to use in project planning — there was little correlation among the data, and the data was not available at the right time (for instance, code size wasn't known until the work was completed). Project managers needed a way to predict and measure what they were planning.

Third level. The key element of the initiative was to be able to predict development effort and duration. We chose two measures to add to those we were already using: (1) size estimates and (2) defect removal for the entire development cycle.

Because the inspection program had been in place since early 1990, we knew we would have significant amounts of data on defect removal. Size estimating was more difficult because we had to move from an effort-based estimating system (sometimes biased by available resources) to one based on quantitative measures that were unfamiliar to most of the staff. The size measures were necessary to derive expected numbers of defects, which then could be used to predict test schedules with greater accuracy. This data also provided feedback to the planning organization for future projects.

To meet the needs of the model shown in Figure 4, we needed the following measures (italics designate changes from the prior list):

- effort in person-months,
- computer resources used,
- *estimated product size at each development stage,*
- product size *after coding,*
- *product size after each test stage,*
- number of defects *found in all development stages from inspections* (in this article, inspection defects refer to major defects), *unit test,* integration test, and system test, and
- *estimated completion date for each phase.*

The sidebar "Data collection sheet" shows a sample form used to compile data.

Project planning

Once the project team develops the first size estimate, the project manager begins to use the data — as well as historical data from our metrics database — for effort and schedule estimating. Several examples from actual projects illustrate these points.

Using defect data to plan test activities. We use the inspection and test defect databases as the primary defect-estimation source. The inspection data provides defect detection rates for design and code by product identifier (PID). Our test database can be searched by the same PID, so a defect depletion curve[2] for the project can be constructed by summarizing all the project's PIDs. (Several interesting examples in Humphrey[2] provided a template for constructing a simple spreadsheet application that we used to plan and track defect injection and removal rates.) Figure 5 shows such a curve for one project. The size and defect density estimates were based on experience from a prior project. The project manager estimated the unit and integration test effort from the defect estimates and the known cost to find and fix defects in test. The estimates and actual amounts are compared in the "Project tracking and analysis" section below.

Data collection sheet

This "Data collection sheet," developed by Kathy Griffith, Software Engineering Process Group project manager at Bull, compiles effort, size, defect, and completion data. Although the sheet is somewhat busy, only six data elements are estimated or collected at each development-cycle phase. The cells with XX in them indicate data collected at the end of high-level design; the cells with YY are derived from the XX data.

DATA COLLECTION SHEET

Project Name □

Build □

Product or Feature Group Identifier(s) (PIDs, IDs, etc.) □ □ □ □ □

Date of Initial Estimates □

	REQ	HLD	LLD	CODE	LEV1	LEV2	LEV3	LEV4	GS	TOTAL
N&C Original KLOC Est										■
N&C Revised KLOC Est		XX								■
N&C KLOC Actuals	■	■	■							■
Effort - Estimate (PM)									■	
Effort - Revised (PM)		XX							■	
Effort - Actual (PM)		XX							■	
# Defects - Estimate		YY								
# Defects - Actual		YY								
Est Phase End Dates		XX								

GS = General Ship
HLD = High-Level Design
LLD = Low-Level Design

LEV1 = Unit, or Level 1, Test
LEV2 = Integration, or Level 2, Test
LEV3 = System, or Level 3, Test
LEV4 = Beta, or Level 4, Test

N&C = New and Changed
PID = Product IDentifier
PM = Person Months
REQ = Requirements Analysis

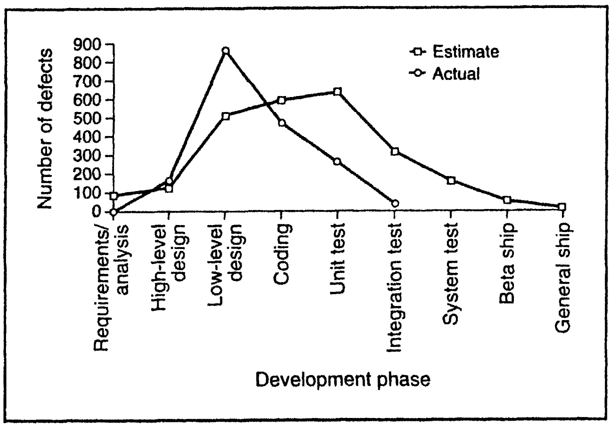


Figure 5. Estimated versus actual defect depletion curves.

Multiple data views. The data in Figure 6 helped the project manager analyze results from the integration test. The project team had little experience with the type of product to be developed, so a large number of defects were predicted. The team also decided to spend more effort on the unit test. After the unit test, the results seemed within plan, as shown in Figure 6a. During the integration test, some concern was raised that the number of defects found was too high. Once the data was normalized against the project size and compared to projections for the number of defects expected by the development team, the level of concern was lowered.

However, this project had a serious requirement error that was discovered in the later stages of the system test; this demonstrated why it's important to look at more than the total number of defects or the defect density, even when the number of defects is below expectations. A closer look at the early development stages shows that very few requirements or high-level design defects were found in the inspections. The low-level design inspections also found fewer defects than expected. What the project members missed in the data analysis during the unit and integration tests was the large number of design errors being detected (see Figure 6b). This example demonstrates the value of independent data collection and analysis as soon as it is available.

An objective analysis, or at least an analysis that looked at the project from a different viewpoint, might have spotted the anomaly. Unit test data accuracy might have been questioned as follows:

- Are some of the errors caused by high-level design defects?
- Why weren't any design defects found in the integration test?

When the data was charted with the defect source added, the design-defect data discovery rate in the unit test was obvious. The inaccuracy of the integration test data also became apparent. A closer look at the project revealed the source of the defect data had not been collected.

We also questioned members of the design inspection teams; we found that key people were not available for the high-level design inspection. As a result, we changed the entry criteria for low-level design to delay the inspection for two to three weeks, if necessary, to let a key system architect participate in the inspection. Part of the change required a risk analysis of the potential for scrubbing the low-level design work started before the inspection.

Using test cost. On one large project, the measured cost in the integration test was much higher than expected. Even though you know the cost of defects in test is high, an accurate cost tally can surprise you. If you haven't gathered the data for a project, the following example may convince you that the effort is worthwhile.

On this large project, it took

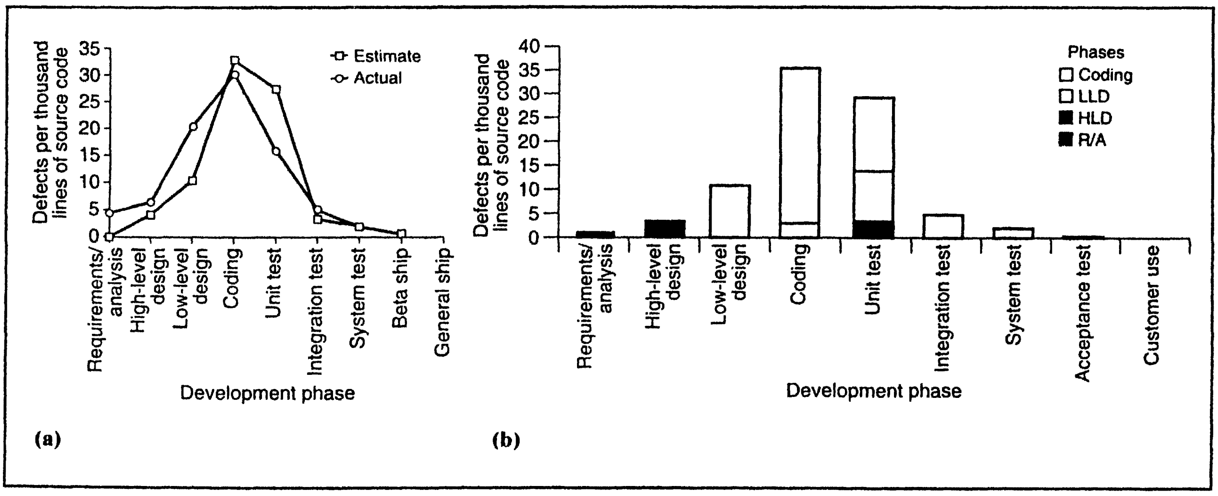


Figure 6. Projected versus actual number of defects found per thousand lines of code from inspections and test (a), and additional information when the defect source is included (b).

- 80 hours to find and fix the typical defect,
- 3.5 person-months to rebuild the product and rerun the test suite, and
- 8-10 calendar days to complete the retest cycle.

Three months' effort represents the fixed cost of test for this product area.

This analysis reemphasizes the need to spend more time and effort on inspecting design and code work products. How many additional hours should be spent inspecting the work products (design, code, and so forth) versus the months of effort expended in test?

Table 1. Defect density inferences.

Defect Density Observation	Inferences
Lower than expected	Size estimate is high (good). Inspection defect detection is low (bad). Work product quality is high (good). Insufficient level of detail in work product (bad).
Higher than expected	Size estimate is low (bad). Work product quality is poor (bad). Inspection defect detection is high (good). Too much detail in work product (good or bad).

Sanity check. Size estimates and software development attribute ratings are used as input to the Cocomo (Constructive Cost Model) estimating model.[3] (Joe Wiechec of Bull's Release Management group developed a Cocomo spreadsheet application based on Boehm[3] for schedule and effort sanity checks.) The accuracy of the effort estimate produced by Cocomo depends on accurate size estimate and software development attribute ratings (analyst capability, programmer capability, and so forth).

We compare the assumed defect rates and cost to remove these defects with the Cocomo output as a sanity check for the estimating process. Since project managers often assign attribute ratings optimistically, the defect data based on project and product history provides a crosscheck with the cost for unit and integration test derived from the Cocomo estimate. Reasonable agreement between Cocomo test-effort estimates and estimates derived from defect density and cost to find and fix per-defect figures confirm that attribute ratings have been reasonably revised. This sanity check works only for the attribute ratings, since both the Cocomo effort estimates and test cost estimates depend on the size estimate.

Project tracking

The keys to good project tracking are defining measurable and countable entities and having a repeatable process for gathering and counting. During the design phase, the metrics available to the project manager are

- effort spent in person-months,
- design-document pages, and
- defects found via work product review, inspection, or use in subsequent development stages.

Interpreting effort variance. When effort expenditures are below plan, the project will typically be behind schedule because the work simply isn't getting done. An alternative explanation might be that the design has been completed, but without the detail level necessary to progress to the next development phase. This merely sets the stage for disaster later.

We use inspection defect data from design inspection to guard against such damaging situations. If the density falls below a lower limit of 0.1 to 0.2 defects per page versus an expected 0.5 to 1.0 defects per page, the possibility increases that the document is incomplete. When defect detection rates are below 0.5 defects per page, the preparation and inspection rates are examined to verify that sufficient time was spent in document inspection. We also calculate the inspection defect density using the number of major defects and the estimated KLOC for the project. If the defect density is lower than expected, either the KLOC estimate is high or the detail level in the work product is insufficient (see Table 1).

When trying to determine which of the eight possible outcomes reflect the project status, project managers must draw on their experience and their team and product knowledge. I believe the project manager's ability to evaluate the team's inspection effectiveness will be better than the team's ability to estimate the code size. In particular, a 2-to-1 increase in detection effectiveness is far less likely than a 2-to-1 error in size estimating.

When effort expenditures are above the plan and work product deliverables are on or behind schedule, the size estimate was clearly lower than it should have been. This also implies later stages will require more effort than was planned.

In both cases, we found that the process instrumentation provided by inspections was very useful in validating successful design completion. The familiar "90 percent done" statements have disappeared. Inspections are a visible, measurable gate that must be passed.

Project tracking and analysis. Inspection defect data adds several dimensions to the project manager's ability to evaluate project progress. Glass[4] and Graham[5] claim that defects will always be a part of the initial development effort. (Glass says design defects are the result of the cognitive/creative design process, and Graham says errors are "inevitable, not sinful, and unintentional.") Humphrey presents data indicating that injection rates of 100 to 200 defects per KLOC of delivered code are not unusual.[6] Although we have seen a 7-to-1 difference in defect injection rates between projects, the variance is much less for similar projects.

For the same team doing similar work, the defect injection rate is nearly the same. The project analyzed in Figure 5 involved a second-generation product developed by many of the people who worked on the first-generation product. At the end of the low-level design phase, Steve Magee, the project manager, noticed a significant difference in the estimated and actual defect-depletion counts for low-level design defects. The estimate was derived from the earlier project, which featured many similar characteristics. There was a significant increase in the actual defect data. We

Table 2. Measures and possible inferences during requirements and design phases.

Measure	Value	Inference
Effort	Above plan	Project is larger than planned, if not ahead of schedule; or project is more complex than planned, if on schedule.
	Below plan	Project is smaller than estimated, if on schedule; or project is behind schedule; or design detail is insufficient, if on or ahead of schedule.
Defects Detected	Above plan	Size of project is larger than planned; or quality is suspect; or inspection detection effectiveness is better than expected.
	Below plan	Inspections are not working as well as expected; or design lacks sufficient content; or size of project is smaller than planned; or quality is better than expected.
Size	Above plan	Marketing changes or project complexity are growing — more resource or time will be needed to complete later stages.
	Below plan	Project is smaller than estimated; or something has been forgotten.

also had some numbers from the first project that suggested inspection effectiveness (defects found by inspection divided by the total number of defects in the work product) was in the 75 percent range for this team.

Again, we were able to use our inspection data to infer several theories that explained the differences. The defect shift from code to low-level design could be attributed to finding defects in the low-level design inspections on the second project, rather than during code inspections in the first project. A closer look at the first project defect descriptions from code inspections revealed that a third of the defects were missing functionality that could be traced to the low-level design, even though many defects had been incorrectly tagged as coding errors. Reevaluating the detailed defect descriptions brought out an important fact: The percentage of design defects detected in code inspection on the first project was higher than we thought.

We were also concerned by the number of defects, which exceeded the total we had estimated even after accounting for the earlier detection. It seemed more likely that the size estimate was low rather than that there was a significant increase in defect detection effectiveness. In fact, the size estimate was about 50 percent low at the beginning of low-level design. The project manager adjusted his size estimates and consequently was better able to predict unit test defects when code inspections were in progress, and time for both unit and integration test.

Using defect data helped the project manager determine that design defects were being discovered earlier and project size was larger than expected. Hence, more coding effort and unit and integration test time would be needed.

Figure 7 shows the actual data as this project entered system test. Comparing the data in Figures 5 and 7 indicates a shift in defect detection to earlier stages in the development cycle; hence, the project team is working more effectively.

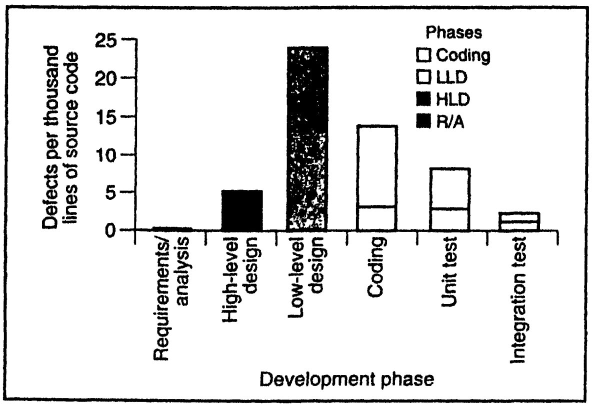


Figure 7. Actual defect density data for the project depicted in Figure 5.

Defect data can be used as a key element to improve project planning. Once a size estimate is available, historical data can be used to estimate the number of defects expected in a project, the development phase where defects will be found, and the cost to remove the defects.

Once the defect-depletion curve for the project is developed, variances from the predictions provide indicators that project managers can examine for potential trouble spots. Table 2 summarizes these measures, their value (above or below plan), and the possible troubles indicated. These measures and those listed in Table 1 answer many of the questions in the design box in Figure 3.

One difficulty project managers must overcome is the unwillingness of the development staff to provide defect data. Grady[7] mentions the concept of public versus private data, particularly regarding inspection-data usage. Unless the project team is comfortable with making this data available to the project manager, it is difficult to gather and analyze the data in time for effective use.

I believe that continuing education on the pervasiveness of defects, and recognition that defects are a normal occurrence in software development, is a critical first step in using defect data more effectively to measure development progress and product quality. Only through collecting and using defect data can we better understand the nature and cause of defects and ultimately improve product quality. ■

Acknowledgments

I thank John T. Harding and Ron Radice for their many hours of discussion on defect removal as a project management tool; Steve Magee, Fred Kuhlman, and Ann Holladay for their willingness to use the methods described in this article on their projects; Jean-Yves LeGoic for his excellent critique; Barbara Ahlstrand, Robin Fulford, and George Mann for inspecting the manuscript; and the anonymous referees for their helpful recommendations.

References

1. E. Weller, "Lessons from Three Years of Inspection Data," *IEEE Software*, Vol. 10, No. 5, Sept. 1993, pp. 38-45.
2. W. Humphrey, *Managing the Software Process*, 1990, Addison-Wesley, Reading, Mass., pp. 352-355.
3. B.W. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, N.J., 1981.
4. R. Glass, "Persistent Software Errors: 10 Years Later," *Proc. First Int'l Software Test, Analysis, and Rev. Conf.*, Software Quality Engineering, Jacksonville, Fla., 1992.
5. D. Graham, "Test Is a Four Letter Word: The Psychology of Defects and Detection," *Proc. First Int'l Software Testing, Analysis, and Rev. Conf.*, 1992, Software Quality Engineering, Jacksonville, Fla.
6. W. Humphrey, "The Personal Software Process Paradigm," *Sixth Software Eng. Process Group Nat'l Meeting*, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, 1994.
7. R. Grady, *Practical Software Metrics For Project Management and Process Improvement*, Prentice Hall, Englewood Cliffs, N.J., 1992, pp. 104-107.

Using Metrics in Management Decision Making

George Stark and Robert C. Durst, Mitre Corporation

C.W. Vowell, NASA Johnson Space Center

The metrics effort within NASA's Mission Operations Directorate has helped managers and engineers make decisions about project readiness by removing the inherent optimism of "engineering judgment."

Over the years, NASA spacecraft and ground systems have become increasingly dependent on software to meet mission objectives. Figure 1 shows the growth in software size for several representative systems over time. The on-board software of unmanned spacecraft has grown from around 500 source lines of code (LOC) in Mariner 9 to an estimated 35,000 LOC for the Cassini spacecraft scheduled for launch in 1997. Software supporting on-board manned systems has grown from 16,500 LOC for the Apollo Saturn V to more than 500,000 LOC for the shuttle, and 900,000 LOC is projected for the space station data-management system. The ground systems used to train the astronauts and to monitor and control the spacecraft contain an average of more than one million LOC. Software supporting the mission control center has quadrupled in size over the last 10 years to more than 3 million executable LOC. For each of these systems, the amount of software has become the dominant factor contributing to increased system complexity. To better understand and manage any risks that might result from this increase in complexity, the Mission Operations Directorate (MOD) initiated a software metrics program in May of 1990.

The key requirement behind the development and implementation of the metrics initiative was to monitor a project's progress unobtrusively. To meet this requirement, the following four environmental criteria were established as essential to the definition of the metrics set. First, the metrics had to be relevant to the MOD development and maintenance environment. That is, they had to be relevant to large, real-time systems that involve multiple organizations and that are coded in multiple languages. Second, collection and analysis of the metrics had to be cost-effective. Third, multiple metrics were required during each reporting period to cross check the indications from any single metric and to provide a complete picture of project status. Fourth, the metrics needed to have a strong basis in industry or government practice for establishing "rule-of-thumb" thresholds for use by project managers.

Reprinted from *Computer*, Vol. 27, No. 9, Sept. 1994, pp. 42–48.

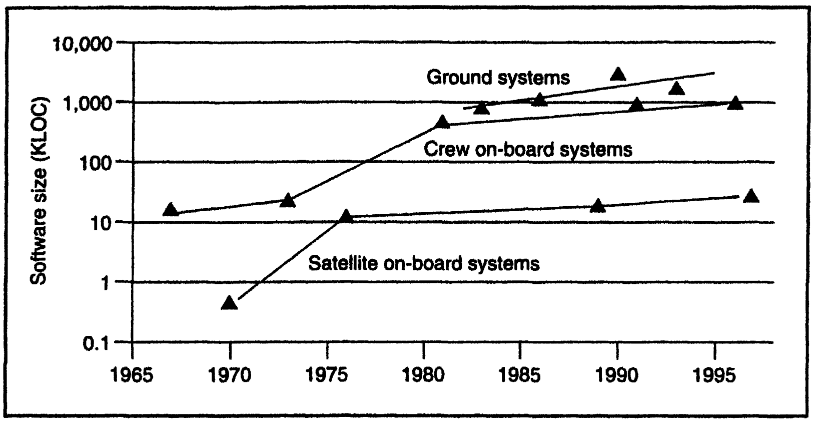


Figure 1. Software size for NASA projects has grown over time.

After defining the environmental criteria, we engaged in a three-step process to implement the software metrics initiative: definition, documentation, and education. For the definition step, we applied Basili's goal/question/metric paradigm[1] and the four environmental criteria to define metrics for software development and maintenance. The Basili technique involves establishing one or more organizational goals, posing questions that address the organization's progress toward meeting those goals, and defining measurements that can be collected to answer the questions. For example, we defined an organizational goal to minimize the effort and schedule used during software maintenance. We posed two questions to address this goal: Where are the resources going? How maintainable is the system?

To answer these questions and still meet the four environmental criteria, we identified the following metrics: software staffing, service request (SR) scheduling, Ada instantiations, and fault type distribution. The software staffing and SR scheduling metrics provide insight into the total resources being applied to the project, while the Ada instantiations and fault type distribution metrics provide insight into the nature of the maintenance workload. (Ada instantiations provide insight into whether generics require repair or merely adaptation, and fault type distribution indicates what types of bugs are being identified and corrected.) To answer the maintainability question, we selected computer resource utilization, software size, fault density, software volatility, and design complexity. Working definitions of these and other metrics are included in the sidebar.

To document the metrics, we developed handbooks containing precise definitions and implementation details for project managers and engineers[2,3] and pulled together a set of stand-alone tools to aid in metrics analysis. Data was collected for six projects over two years to

Mission Operations Directorate software metrics descriptions

Ada instantiations — The size and number of generic subprograms developed and the number of times they are used within a project.

Break/fix ratio — Number of DRs resulting from a discrepancy report (DR) fix or a service request (SR) change divided by the total number of closed DRs + SRs over the same time period.

Computer resource utilization (CRU) — The percentage of CPU, memory, network, and disk utilization.

Design complexity — The number of modules with a complexity greater than an established threshold.

Development progress — The number of modules successfully completed from subsystem functional design through unit test.

Discrepancy report (DR) or service request (SR) open duration — The time lag from problem report or service request initiation to closure. A discrepancy report is a change made to software to correct a defect. A service request is a change made to software to add or enhance a capability.

DR/SR closure — Actual DR or SR receipts and closures by (sub)system by month.

Fault density — The open and total defect density (DRs normalized by software size) over time.

Fault type distribution — Percentage of defects closed with a software fix by type of fault (for example, logic, error handling, standards, interface).

Maintenance staff utilization — Engineering months per SR and per DR written by (sub)system.

Requirements stability — The trend of the total number of requirements to be implemented for the project over time.

Software reliability — The probability that the software "works" for a specified time under specified conditions.

Software size — The number of lines of code in the system that must be tested and maintained.

Software staffing — The number of software engineering and first-line management personnel directly involved with the software.

Software volatility — Percentage of modules changed per release.

SR scheduling — The length of time it takes to close an SR that requires a software change and the amount of engineering effort spent on SR closures.

Test case completion — The percentage of test cases successfully completed.

Test focus — The percentage of problem reports closed with a software fix.

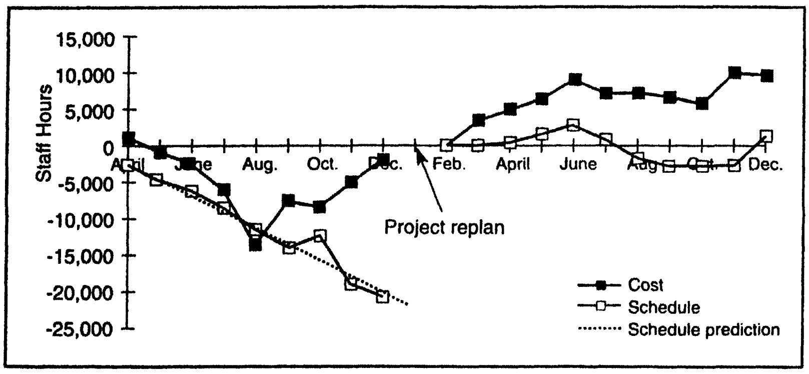


Figure 2. An earned-value chart for a MOD project helped keep cost and schedule on track.

test the process and to provide material for subsequent training. Finally, an education program was developed to ensure a common understanding of the metrics and their application. All MOD development and contractor personnel are trained using data from previous projects.

The following sections describe four specific decisions that were made using metrics data collected on MOD projects. We also provide a description of the metric toolkit and present some of our observations regarding metric data analysis and its real-world use.

Deciding to replan a project

The problem: How can I tell if my project is on schedule and within budget? MOD uses *earned value* to help answer this question. Earned value is a technique that combines the development progress metric, the staffing metric, and the expected cost of the project. A number of clearly identifiable development progress milestones are established, and a percentage of the cost and the expected number of staff-hours per calendar time to complete the task is allocated to each milestone. (Cost is tracked by staff-hours to protect contractor rate confidentiality.) The staff-hours value is "credited" only when the milestone is 100 percent complete. The total earned value equals the new milestones completed plus the prior completed milestones. Each month the earned value is reviewed. When enough data points are available, a linear extrapolation of the project schedule and cost is calculated. Management then takes corrective action if there is a definite negative trend to the data.

Figure 2 shows an example earned-value graph by month for a project. This project began in April without earned-value measurement. Its scheduled completion date was October of the following year. In August, the project was baselined with earned value. The project was 13,200 hours behind schedule and 11,000 hours over budget (negative numbers indicate over budget or behind schedule). Because this was the first earned-value project in MOD, a decision to rebaseline was not made immediately. Instead, we gathered further metrics data and conducted supporting interviews with the project team.

By December, it was clear that action was required: The project was 20,000 hours behind schedule and over budget by almost 1,000 hours. (A reallocation of staffing resources in August accounted for the progress toward meeting budget, but the schedule continued to slip.) The dashed line in Figure 2 is a forecast based on the equation $y = mx + b$, where y is the hours earned and x is the month (that is, April = 1, May = 2, . . . , December = 9). The values for the slope (m) and the intercept (b) were calculated to be $m = -2{,}197$ and $b = 40$. Note that the slope is in units of staff-hours per month. Thus, assuming that there are 2,000 staff-hours in a staff-year, this slope indicates that the project schedule is slipping at a rate of more than one staff-year each month.

A review of the two most problematic subsystems showed some "requirements creep" and some staffing problems. The managers decided to review the requirements and replan the project. The review and replan took one month; no original requirements were scrubbed. The new plan mitigated the requirements creep by incorporating the use of common software and commercial products available on the newly procured target platform. The new plan moved a small number of staff from maintenance to development and extended the original delivery date by two months. In October of the second year, a minor correction was made to the plan. As shown in the graph, this new plan delivered the system with the required functionality under budget (by 9,827 hours) and on schedule (1,500 hours ahead). Metrics collection made the problems visible, and the subsequent analysis drove these corrective decisions. A rough estimate of the cost avoidance from these decisions is calculated as (predicted schedule slip – actual schedule slip) × staff (or 9 months – 2 months) × 110 staff = 770 staff months.

Deciding to maintain or redesign software

The problem: How hard will it be for another organization to maintain this software? One approach to this question is to find relationships between the measurable characteristics of programs and the difficulty of maintenance tasks. Researchers have developed measures of software complexity that can be used to understand the structure of a software module. More than 100 complexity measurements have been proposed in the literature, but McCabe's cyclomatic complexity metric[4] is studied and used most often. McCabe defines the complexity of a program on the basis of its structure, using the number and arrangement of decision statements within the code. McCabe complexity is calculated as the number of decisions in the code plus one. Myers extended the metric to include predicates (for example, AND and OR) and decision-making statements in the calculation.[5] Because predicates do indeed create additional independent paths through a module, this metric is more comprehensive than its forerunner and is used within MOD.

Some researchers have expressed doubts about the validity of complexity measurement. These doubts have revolved around predictive models using the metrics and around the experimental designs used in some validation studies. Some practitioners, however, have found complexity measurement useful in planning for and assessing software development risks, in allocating resources during testing, and in managing maintenance efforts. Complexity measurement is implementable as part of a project's coding standard, since inexpensive tools are available to compute the values and report exceptions to the standard. The Software Engineering Laboratory at NASA's Goddard Space Flight Center has successfully used complexity measurement on a number of projects. The Safety Reliability and Quality Assurance organization at Johnson Space Center (JSC) has also applied the concepts successfully.

Within MOD, 16 systems currently in maintenance were analyzed. Seven of these systems were coded in C, four in Ada, and five in Fortran. For a given system, each software procedure was analyzed, and a cumulative distribution function was generated. There were typically several hundred to several thousand procedures in a system. Figure 3 shows the cumulative distribution functions of the extended McCabe complexity measure for 16 systems. (Note that the figure is a logarithmic scale on the *x*-axis.)

The figure indicates that 50 percent of system A functions had a complexity of less than or equal to 10, and 90 percent were less than or equal to 80. Based on published rules of thumb (for example, procedures should have McCabe complexity of less than 10) and the noticeable gap between the A and B systems and the other 14 systems, managers at JSC considered these two systems risk areas. They conducted a further investigation of these systems to determine the number of problem reports written since release, the number of users, and maintenance staff size. Management decided to retire system A and to find another approach to implement its function. An evaluation of commercial off-the-shelf products turned up a candidate that met more than 80 percent of system A functionality. This product has since been implemented and is now used by a majority of the users. Management decided to accept the risk on system B, since it was relatively error free and was developed, maintained, and used by a single organization. This result has allowed management to reallocate system A maintenance staff and thus concentrate additional effort in other areas.

For the systems in Figure 3, there was no statistical difference between languages in terms of complexity. That is, the C systems studied were no more complex than the Ada or the Fortran systems. Thus, management used this graph to define categories of complexity. Figure 4 is a decision chart for use on new projects. Systems with distributions in the upper left-hand section of the chart are considered to have straightforward logic, systems that fall in the middle area are considered "standard complexity," and systems that fall to the right side of the chart are considered to be too logically complex. When systems cross the boundary from standard to logically complex, the programming team is encouraged to review those modules. The logically complex systems are candidates for further review and possible reengineering. We are currently evaluating other techniques of complexity measurement (for example, Munson's relative complexity measure), but at this stage of the metrics program the extended McCabe metric is helping MOD managers make more informed decisions.

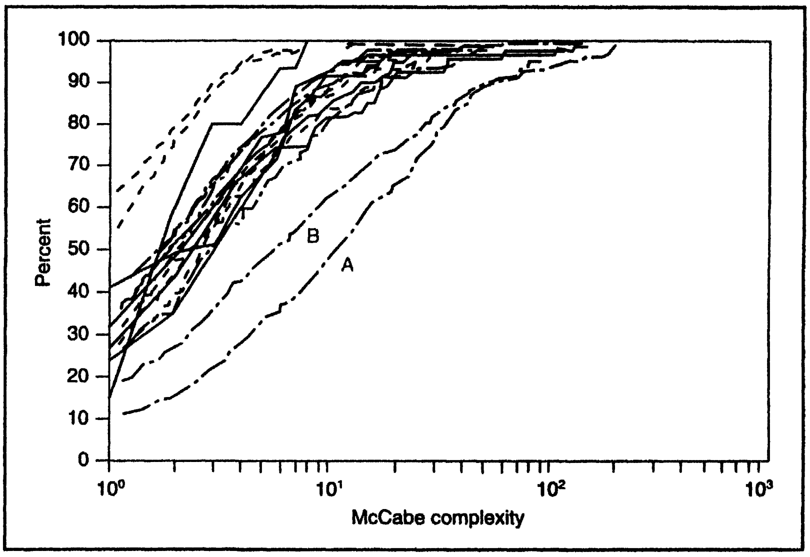


Figure 3. Cumulative distributions of the McCabe complexity measure revealed systems A and B as risk areas.

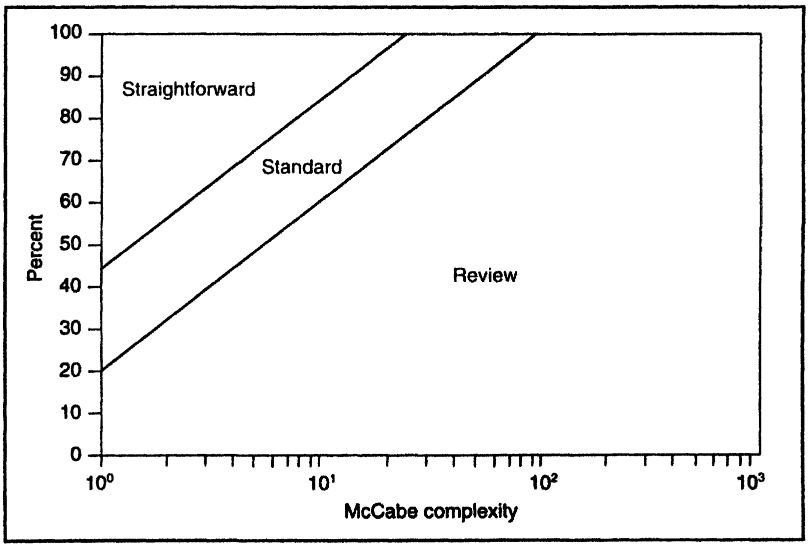


Figure 4. A McCabe complexity decision chart showing categories of complexity.

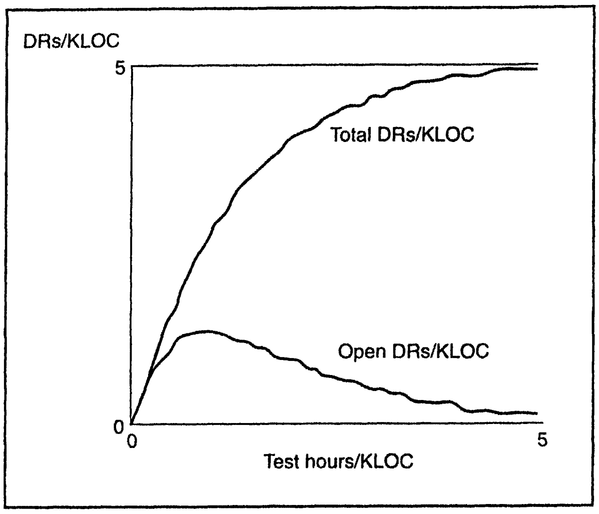


Figure 5. An ideal profile from subsystem defect-density testing.

Deciding when to integrate subsystems

The problem: How can I tell when my subsystems are ready to be integrated? We use the defect density metric to quantify the relative stability of a software subsystem and to identify any testing bottlenecks or possible overtesting by examining the fault density of the subsystem over time. There are three components to this metric: Total Density (*T*), Open Density (*O*), and Test Hours (*H*). The term *density* implies normalization by LOC. In practice,

T = Total number of software defects charged to a subsystem/KLOC,

O = Number of currently open subsystem software defects/KLOC, and

H = Active test hours per subsystem/KLOC.

The metric is then tracked as a plot of *T* and *O* versus *H*. A graph of *T* versus *H* indicates testing adequacy and code quality. Many problems are discovered early in the subsystem's testing phase, with increasingly fewer problems found as testing proceeds. Thus, the ideal graph begins with a near infinite slope and, as testing and debugging continues, approaches a zero slope. If the slope doesn't begin to approach zero, a low-quality subsystem or inefficient testing is indicated and should be investigated. The plot of *O* versus *H* indicates the problem-report closure rate. Again, more problems should be backlogged at the beginning of test; then, as debuggers begin to correct the problems, the slope should become negative, indicating that the debuggers are working off their backlog. If the slope of the *O*-versus-*H* curve remains positive, it means the testers are finding faults faster than the debuggers can resolve them; the remedy is to halt testing until a new release can be delivered and the backlog of faults is reduced. An ideal defect density plot is shown in Figure 5.

Rules of thumb for this metric depend on the development environment and the organizational processes. The MOD rule of thumb is that total discrepancy reports (DRs) should be in the 5-per-KLOC range, with values between 3 and 10 considered normal. The test hours should be in the 2-per-KLOC range, with values between 1 and 10 considered normal. In general, the shape of the curves indicates the relative subsystem stability within a project. That is, by comparing the curves, one can determine if the subsystems are ready for integration. Total defects per KLOC should flatten out over time, and open defects per KLOC should decrease and approach zero. Too few defects or too few test hours may indicate poor test coverage or unusually high code quality, while too many of either may indicate poor code quality.

Data for the defect density metric is plotted as both total defects and open defects per KLOC versus test hours per KLOC by subsystem. On this project, we reviewed subsystems with increasing numbers of open defects, low test hours per KLOC, or no flattening of total defects, because any of these conditions indicate a risk to successful deployment of the system.

Figures 6 and 7 plot defect density for two contrasting subsystems. Figure 6 matches the expected curves of Figure 5, indicating that subsystem E is mature and ready for integration. Comparison of subsystem H in Figure 7 with the expected profile shows differences that can be interpreted as risk signals indicating that the developers are having difficulty — that is, the total defect density is increasing, the open density is not decreasing, and very few test hours per KLOC have been expended. (The straight drop in defects/KLOC occurred when more code was added to the subsystem during test, as indicated by the software size metric.) Further, comparison with the other subsystems (for example, subsystem E from Figure 6) shows unequal testing and debugging of this subsystem. It is not ready for integration.

Based on this curve, MOD managers decided to allocate an additional 10 days of stand-alone testing for subsystem H. This decision lengthened the overall delivery schedule by three days, but without this additional testing, the integration process and the system-level testing would likely have been more difficult and time consuming. Note that analysis of this kind is not objective, nor is it completely quantitative. No universal thresholds or algorithms for subsystem acceptance are defined. However, we believe that the quantitative data provides support for higher quality subjective decision making.

Deciding whether a test schedule is reasonable

The problem: How can I tell if a test schedule makes sense? One way is to use historical data from previous projects as a reference. One can even account for "lessons learned" from previous projects in the analysis. The following example is from a project that immediately followed one that experienced significant code growth. In planning the follow-on project, the developers recalled that they had underestimated code size and therefore underestimated their test time requirements. The contractor responded by assuming similar code growth as before and increasing the planned number of test hours/KLOC. Unfortunately, this

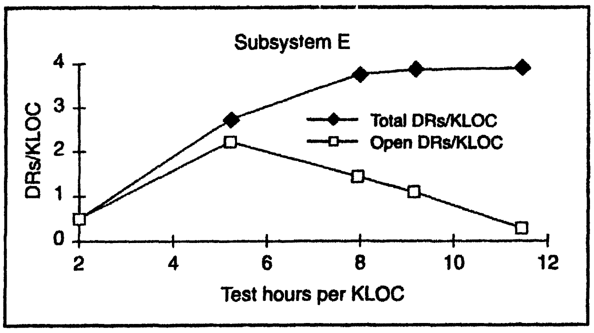


Figure 6. A subsystem defect-density profile that indicates there are no problems.

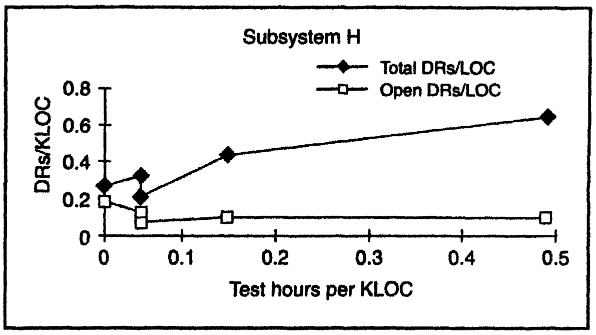


Figure 7. A subsystem defect-density profile indicating that corrective action is required.

was a major expense item and affected the product availability date. MOD wanted to take a different approach: control code growth and determine the test schedule.

The initial contractor schedules were available approximately one year before the start of testing. We performed a "sanity check" on the schedule and estimated a lower bound for the test schedule by factoring out the assumed code growth. (We divided the number of test hours requested by the code growth factor experienced on the previous project — LOC delivered/LOC planned.) We then estimated the months in test using the previous project's test-hour delivery rate (that is, test hours/calendar months). This yielded an estimate of 2.4 calendar months rather than the initial request of 6 months. Next, we estimated an upper bound for the test schedule by estimating the time required to deliver the full number of test hours based on the previous project's delivery rate. This resulted in an estimate of 7 months.

Management believed they could control requirements growth and other factors contributing to code growth so that the system would not experience the same growth as the previous delivery. To be conservative, they allowed for a growth factor of two, resulting in a 5-month test schedule. This reduced the contractor's schedule by one month, for an estimated savings of $200,000. Management reinforced the use of metrics by using metrics to closely monitor actuals versus plans on a monthly basis. By using the metrics as early indicators of changes in the project's status, management was able to hold code growth to a factor of 1.67. As a result, the project completed testing in slightly over 4 months, saving an estimated $300,000.

Metrics toolkit

Two characteristics that facilitated making the decisions described in the previous section were the ease of analysis and the consistency in the data collection. To ensure that these characteristics were present in the metrics program, we defined a standard suite of tools for use by analysts and project personnel. The toolkit supports analysis of the metrics to answer the questions and meet the goals that originally generated the metric set. Recall that answering a single question can require analysis of many metrics. Thus, it is important for the toolkit to collect and integrate data from multiple metrics. In our environment, the toolkit required a data repository element (for example, a database or spreadsheet), a cost/resource estimation tool, a size/complexity collection tool, and a reliability estimation tool.

The data repository element is based on a spreadsheet running on a desktop computer. The spreadsheet lets managers track planned and actual measurements. It also performs simple linear regression on individual metrics and provides graphical displays of the metrics data.

Additional cost and schedule information is provided by JSC's CostModeler[6] (an upgrade of JSC's CostModl). This tool helps a manager estimate the effort, cost, and schedule required to develop and maintain computer systems. CostModeler does not directly implement any specific cost estimation algorithm but instead lets users tailor appropriate models for their needs. It includes implementations of the JSC KISS (Keep it simple, stupid) model and four variations of Boehm's Cocomo (constructive cost model).

Another tool in the kit is Set Laboratories' UX-Metric, a commercial off-the-shelf code analysis tool that helps engineers evaluate software complexity.[7] The tool supports several measures of complexity, including extended cyclomatic complexity, lines of code, span of variable reference, depth of nesting, number of interfaces, and Halstead software science measures.

Finally, we use the Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) tool for software reliability assessments. This tool is a public domain program available from the Naval Surface Warfare Center in Dahlgren, Virginia.[8] It implements 10 time-domain software reliability models, including those recommended by the American Institute of Aeronautics and Astronautics.[9] This tool allows analysts to forecast test durations, predict field failure rates based on test data, and track operational failure rates.

Although this toolkit is neither completely automated nor integrated in the sense that a user has a single entry point, all the components can share data. MOD project managers have found it useful for the metrics collection and analysis described above. The toolkit costs less than $1,000 and took less than a month to integrate and begin using.

Metrics, models, and decision making

Occasionally, we need to be reminded of the real meaning and use of metrics in our environment. The goal is to create a dialogue between managers and developers or between customers and suppliers. In general, the predictions made from the data have no element of statistical confidence. They are first and foremost design tools used to compare plans to actual results, to identify overly complicated parts, and to serve as input to the system's risk management.

The metrics and the models that use them need not be exact to be useful. Exact models are often so cumbersome and intractable that they are worthless to managers making decisions. Excellent and useful results are regularly obtained from simple models like those described above. Sometimes, however, such models give results that don't make sense in the environment. For example, in our original software reliability estimates we used a model that assumed that the number of lines of code in test was stable. The predictions from this model were not matching the observed failure rates in the project. Upon investigation we found that the size of the system under test was increasing incrementally. We changed to a model that accommodated software growth during test. The new model more closely matched the observed data, and the predictions from the model were useful for management decision making.

The point is that when the results from a model are not reasonable, these results are telling us something and need to be carefully examined: Either we've made a mistake in the mathematics, or the assumptions that define the model are bad. Sometimes an incorrect assumption is hard to notice. The model must be worked backward, manipulating the assumptions to give the best results, or a new approach must be taken.

Thus, metrics analysis begins with insight into the workings of the software development or maintenance processes. It continues with calculations from the conceptual models that reflect that insight. It results in answers to the original questions — answers that may affect decisions and change processes. Those portions of the metrics task that are computationally intensive, such as model execution, are best left to support tools such as those described in the toolkit. Some portions of the metrics task are best done by people. These are the portions that involve insight into the internal workings of the organization's processes. The insights gained by performing a Pareto analysis or debugging a model can result in improvements to the subject process as well as the model of that process. Thus, there is value to the organization in performing some of the metrics task "by hand." Remember, system integrity cannot be achieved without sound engineering applied to established software development and maintenance tasks.

The amount of code in NASA systems has continued to grow over the past 30 years. This growth brings with it the increased risk of system failure caused by software. Thus, managing the risks inherent in software development and maintenance is becoming a highly visible and important field. The metrics effort within MOD has helped the managers and engineers better understand their processes and products. The toolkit helps ensure consistent data collection across projects and increases the number and types of analysis options available to project personnel. The decisions made on the basis of metrics analysis have helped project engineers make decisions about project and mission readiness by removing the inherent optimism of "engineering judgment." ■

References

1. V. Basili and H.D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proc. Ninth Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., Order No. 767 (microfiche only), 1987, pp. 345-357.
2. NASA Johnson Space Center, *DA3 Software Development Metrics Handbook*, Version 2.1, JSC-25519, Houston, Texas, 1992.
3. NASA Johnson Space Center, *DA3 Software Sustaining Engineering Metrics Handbook*, Version 1.0, JSC-26010, Houston, Texas, 1992.
4. T.J. McCabe, "A Complexity Measure," *Trans. on Software Eng.*, Vol. SE-2, No. 4, 1976, pp. 308-320.
5. G. J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity," *SIG-Plan Notices*, Vol. 12, No. 10, 1977, pp. 61-64.
6. B. Roush and R. Phillips, *CostModeler Version 1.0 User's Guide*, Software Development Branch, NASA Johnson Space Center, Houston, Texas, 1993.
7. SET Laboratories Inc., *UX-Metric*, Mulino, Ore., 1990.
8. W.H. Farr and O.D. Smith, "Statistical Modeling and Estimation of Reliability Functions for Software (SMERFS) User's Guide," NSWC TR 84-373 Rev 1, Naval Surface Warfare Center, Silver Spring, Md., 1988.
9. American Institute of Aeronautics and Astronautics, *Recommended Practice for Software Reliability*, ANSI/AIAA R-013-1992, Washington, D.C., 1993.