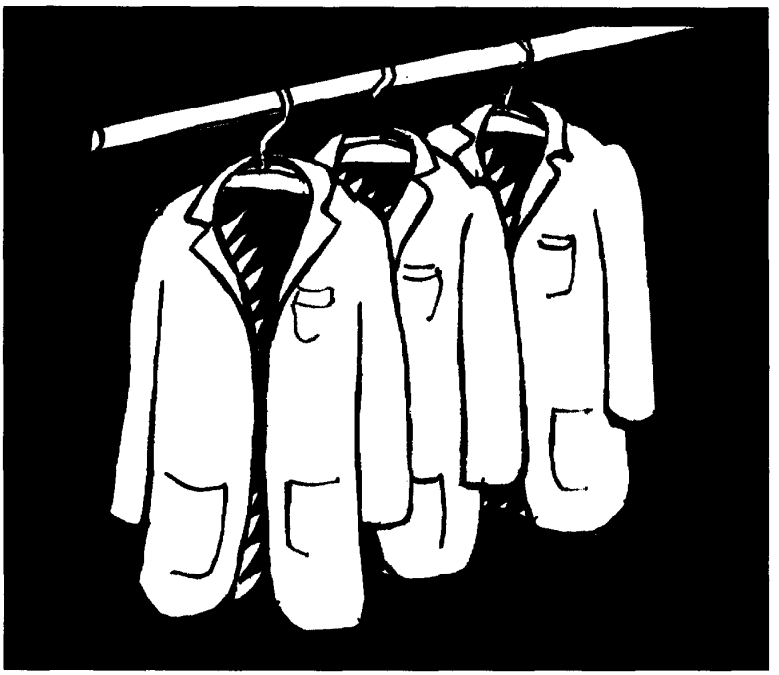
FEATURE

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 programming,
- 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

0740-7459/94/$04.00 © 1994 IEEE

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 mathe-

matical 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.

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.

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

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.

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.

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

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.

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.

TABLE 1
COMPARISON OF TESTING EFFICIENCY

Testing type	Efficiency (defects found per hour)
Regular use	0.210
Black box	0.282
White box.322	
Reading inspections	1.057

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-

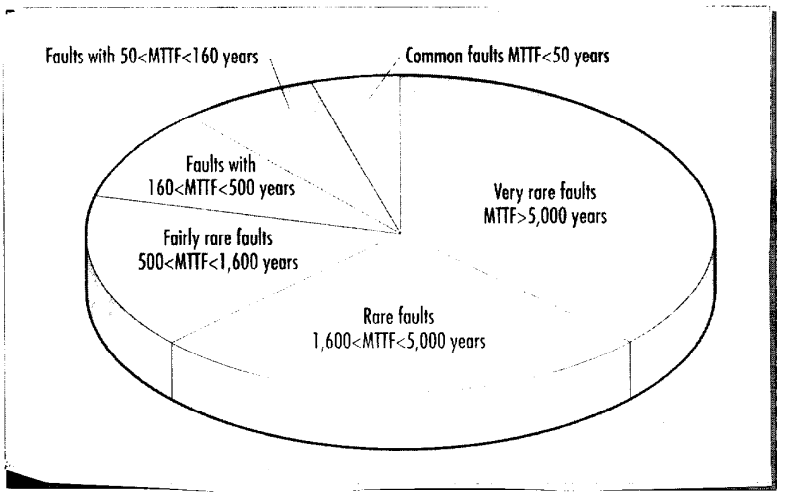


Figure 1. *The relationship between faults and failures, which shows that focusing on faults instead of failures can be fatal. Studies that compare testing methods by using faults may be inappropriate and misleading.*

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.

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!

THERE ARE FAR TOO FEW EXAMPLES OF MODERATELY EFFECTIVE RESEARCH.

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.

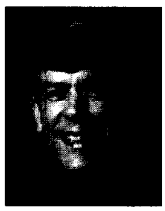
By taking these steps, the entire community should benefit. Finding willing industrial partners for research should be made easier, as the potential benefit to all participants is clear. The European Community has recognized the urgent need for quantitative evaluation performed by industry-research partnerships. 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. ◆

Norman Fenton is a professor of computing science in the Centre for Software Reliability at City University, a consultant on metrics programs, and the leader of SMARTIE, a project to develop a measurement-based framework to assess software-engineering standards and methods. His research interests include software measurement and formal methods, and he has written three books and many papers on these subjects. He is the editor of the Chapman and Hall Computer Science Research and Practice Series and serves on the editorial board of *Software Quality Journal*.

Fenton received a PhD in mathematics from Sheffield University. He is a member of the IEE (chartered engineer), an associate fellow of the Institute of Mathematics and its Applications, and a member of the IEEE Computer Society.

Shari Lawrence Pfleeger is president of Systems/Software, Inc., a consultancy on software engineering, process improvement, and measurement, and a visiting professorial research fellow at both the City University of London's Centre for Software Reliability and the University of North London, where she is evaluating the extent and effect of standards and writing experimentation and case-study guidelines for software engineers. She has written two books on computer science and software engineering and many research papers in computer science and mathematics, and she serves on the editorial board of *IEEE Software* and the advisory board of *IEEE Spectrum*.

Pfleeger received a PhD in information technology and engineering from George Mason University. She is a member of the IEEE, the IEEE Computer Society, and ACM, and the founder of the ACM Committee on the Status of Women and Minorities.



Robert L. Glass is publisher of *The Software Practitioner*, editor of the *Journal of Systems and Software*, a regular columnist for *System Development*, and a visiting professor of software engineering at Linköping University. He is interested in all facets of software engineering, especially in quality and maintenance. He has written 17 books and more than 30 papers on computing and software.

Glass received an MSc in mathematics from the University of Wisconsin at Madison. He is a member of the IEEE, the IEEE Computer Society, and ACM.

Address questions about this article to Fenton or Pfleeger at CSR, City University, 2 Riverside Close, Kingston Upon Thames, Surrey KT1,2JF, England; n.e.fenton or shari@csr.city.ac.uk, or to Glass at Computer Trends, PO Box 213, State College, PA 16804.

Free report from Peter Coad reveals amazing industry breakthrough!

"Object modeling and C++ programming, side-by-side, always up-to-date."

Big CASE tool vendors caught with their pants down!

What if you could have your OOA/OOD model and all of your C++ code continuously up-to-date, all the time, throughout your development effort?

Consider the possibilities...

In one window, you see an object model, with automatic, semi-automatic, and manual layout modes, plus complete view management. Side-by-side, in the other window, you see fully-parsed C++ code. You edit one window or the other. Press a key. Both windows agree with each other. **Together.**

Suppose that you are working on a project with some existing code. (That's no surprise, who'd consider developing in C++ without some off-the-shelf classes?) You read the code in. Hit a button. And seconds later, you see an object model, automatically laid out and ready for you to study side-by-side with the C++ code itself. **Together.**

Or suppose you are building software with other people (that's no surprise either). You collaborate with others and develop software with a lot less hassle, because the fully integrated configuration management feature helps you keep it all...**Together.**

The name of this product? It's earned the name...

Together/C++
continuously up-to-date
object modeling and C++ programming

Call now for your free report!

Key features:
- Continuously up-to-date object modeling & C++ programming
- Automatic, semi-automatic, and manual layout of object models
- Object modeling view management, including view control by C++ construct, regular expression, proximity, layer, or directory
- Fully flexible documentation generation, version control, and SQL generation

"State-of-the-art application development."
-- *Computerworld*/Germany

"You've really hit the nail on the head when it comes to reverse engineering existing C++ code. No other tool comes close to the power and capability of Together/C++."
-- Russell Rudduck, Perot Systems

Money-back gurantee. Purchase Together/C++ and try it out risk-free for 30 days. If for any reason you aren't satisfied, return it for a full refund. (No hassles, no hard feelings either.)We're that confident about Together/C++. You see, Together/C++ has already helped software developers deliver better systems, with success stories in telecommunications, insurance and natural resource managment.

How to order. Order Together/C++ by purchase order, check, or credit card, or for more information, please contact:

Object International, Inc.
Education - Tools - Consulting
8140 N. MoPac 4-200
Austin TX 78759 USA
1-800-OOA-2-OOP
1-512-795-0202 - fax 795-0332
e-mail: object@acm.org

Outside of North America, contact:
Object Int'l Ltd.
Eduard-Pfeiffer-Str.73
D-70192 Stuttgart, Germany
++49-711-225-740 - fax 299-1032
100034.1370@compuserve.com

©1994 Object Int'l, Inc.
All rights reserved.
"Together" is a trademark of Object Int'l, Inc.

IEEE794