Software-Reliability Engineering: Technology for the 1990s

John D. Musa *and* ***William W. Everett****, AT&T Bell Laboratories*

Software engineering is about to reach a new stage — the reliability stage — that stresses customers' operational needs. Software-reliability engineering will make this stage possible.

Where will software engineering head in the 1990s? Not an easy question to answer, but a look at its history and recent evolution may offer some clues. Software engineering has evolved through several stages during its history, each adding a new expectation on the part of users:

• In the initial *functional* stage, functions that had been done manually were automated. The return on investment in automation was so large that *providing* the automated functions was all that mattered.

• The *schedule* stage followed. By this point, users' consciousness had been raised about the financial effects of having important operational capabilities delivered earlier rather than later. The need to introduce new systems and features on an orderly basis had become evident. Users were painfully aware of operational disruptions caused by late deliveries. Schedule-estimation and -management technology used for hardware systems was coupled with data and experience gained from software development and applied.

• The *cost* stage reflected the widespread use of personal computers, where price was a particularly important factor. Technology for estimating software productivity and cost and for engineering and managing it to some degree — however imperfect — was developed.

• We are now seeing the start of the fourth stage —*reliability*— which is based on engineering the level of reliability to be provided for a software-based system. It derives from the increasingly absolute operational dependence of most users on their information systems and the concomitant heavily increasing costs of failure. This stage must respond to the need to consider reliability as one of a set of factors (principally functionality, delivery date, cost, and reliability) that customers view as comprising quality. It must contend with the reality — not always fully recognized by users — that for any stage of tech-

0740-7459/90/1100/0036/$01.00 © 1990 IEEE

nology, improving one of the quality factors may adversely affect one of the others.

In response to this challenge, a substantial technology — software-reliability engineering — has been developed[1] and is already seeing practical use.[2] Trends indicate that it will be extensively applied and perfected in the 1990s.

What is it?

Software-reliability engineering is the applied science of predicting, measuring, and managing the reliability of software-based systems to maximize customer satisfaction. Reliability is the probability of failure-free operation for a specified period. Software-reliability engineering helps a product gain a competitive edge by satisfying customer needs more precisely and thus more efficiently.

Software-reliability engineering includes such activities as

- helping select the mix of principal quality factors (reliability, cost, and availability date of new features) that maximize customer satisfaction,
- establishing an operational (frequency of use) profile for the system's functions,
- guiding selection of product architecture and efficient design of the development process to meet the reliability objective,
- predicting reliability from the characteristics of both the product and the development process or estimating it from failure data in test, based on models and expected use,
- managing the development process to meet the reliability objective,
- measuring reliability in operation,
- managing the effects of software modification on customer operation with reliability measures,
- using reliability measures to guide development process improvement, and
- using reliability measures to guide software acquisition.

Software-reliability engineering works in concert with fault-tolerance, fault-avoidance, and fault-removal technologies, as well as with failure-modes and -effects analysis and fault-tree analysis. It should be applied in parallel with hardware-reliability engineering, since customers are interested in the reliability of the whole *system*.[3]

Why is it important?

The intense international competition in almost all industries that developed in the late 1980s and that will probably increase in this decade has made software-reliability engineering an important technology for the 1990s. The sharply dropping costs of transportation and, particularly, communication, the rapidly increasing competitiveness of the Pacific Rim countries, the ascent of the European Economic Community, and the entry of the Communist bloc into the world economy all indicate the likely continued strength of this trend.

There appears to be a strong correlation between interest in adopting reliability-engineering technology and the competitiveness of the organization or project concerned.

The fact that the world economy has evolved from being labor-intensive to capital-intensive (with the Industrial Revolution) and now to information-intensive to meet this competition makes high-quality information processing critical to the viability of every institution. Thus, customers of software suppliers have become very demanding in their requirements. Fierce competition has sprung up between suppliers for their business.

It is no longer possible to tolerate one-dimensional conservatism when engineering quality factors. If you err on the safe side and build unneeded reliability into products, a competitor will offer your customers the reliability they do need at a lower price or with faster delivery or some combination of both. You must understand and deal with the real interactive multidimensionality of customer needs and make trade-offs. Above all, you must precisely specify, measure, and control the key quality factors.

We have observed an interesting phenomenon: There appears to be a strong correlation between interest in adopting reliability-engineering technology and the competitiveness of the organization or project concerned.

The arguments for achieving quality in software products are persuasive. A study[4] of factors influencing long-term return on investment showed that the top third of companies in customer-perceived quality averaged a return of 29 percent; the bottom third, 14 percent. Increased demand for quality also increases the importance of precisely measuring how well your competitors are doing in providing quality to the market.

It is also becoming increasingly important that the right level of reliability be achieved the first time around. In the past, without customer-oriented measures to guide us, we often approached reliability incrementally. We guessed what the customer wanted, provided an approximation to it, awaited the customer's dissatisfactions, and then tried to ameliorate them.

But the costs of operational disruption and recovery that the customer encounters due to failures have increased and are increasing relative to other costs. With

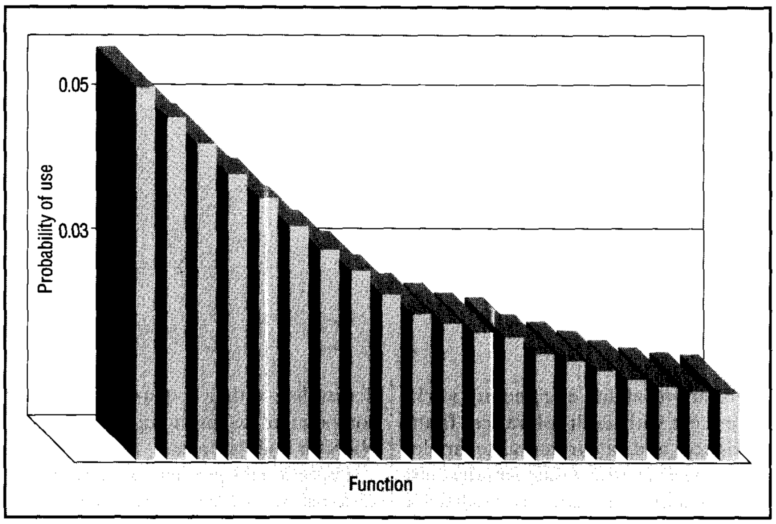


Figure 1. Operational profile.

such pressures, loss of confidence in a software supplier can be rapid. This trend argues for greatly increased and quantified communication between supplier and customer — from the start.

Thus, the application and perfection of sofware reliability engineering is rapidly growing in importance. It provides a measurement dimension of quality that has until recently been lacking, with very detrimental results.

Basic concepts

To understand software-reliability engineering, you must start with some important definitions:

- A software *failure* is some behavior of the executing program that does not meet the customer's operational requirements.
- A software *fault* is a defect in the code that may cause a failure.

For example, if a command entered at a workstation does not result in the requisite display appearing, you have a failure. The cause of that failure may be a fault consisting of an incorrect argument in a calling statement to a subroutine.

Failures are customer-oriented; faults are developer-oriented.

New thinking in software-reliability engineering lets you combine functionality and reliability into one generic reliability figure, if desired. System requirements are interpreted both in terms of functions explicitly included in a product and in terms of those that the customer needs but that are not included in the product. If a function is lacking, the product is charged with a failure each time the customer needs the absent function. Thus a low level of functionality will result in a low level of reliability.

Of course, some objective basis for "needs" must be established, such as an analysis of the customer's operation or the union of all features of competitive products. Otherwise, the needs list could become an open-ended list of unrealistic wishes.

There are two alternative ways of expressing the software-reliability concept, which are related by a simple formula[1]:

- *Software reliability* proper is the probability of failure-free operation for a specified time duration.
- The other expression is *failure intensity*, the number of failures experienced per time period. Time is *execution time*: the actual time the processor is executing the program. As an example, a program with a reliability of 0.92 would operate without failure for 10 hours of execution with a probability of 0.92. Alternatively, you could say that the failure intensity is eight failures per thousand hours of execution.

Varying operational effects of failures can be handled by classifying failures by *severity*, usually measured by threat to human life or by economic impact. You can determine a reliability or failure intensity for each severity class, or you can combine the classes through appropriate weighting to yield a loss function like dollars per hour of operation.

Fault measures, often expressed as *fault density* or faults per thousand executable source lines, are generally not useful to customers. Although they can help developers probe the development process to try to understand the factors that influence it, you must relate them to measures like failure intensity and reliability if — as is proper — measures of customer satisfaction are to be the ultimate arbiters.

Another important concept is the *operational profile*. It is the set of the functions the software can perform with their probabilities of occurrence. Figure 1 shows a sample operational profile. The profile expresses how the customer uses or expects to use the software.[1,5]

Setting improvement goals. Companies' software-improvement goals have usually not been set from the customer's perspective. However, percentage-improvement goals stated in terms of the customer-oriented failure intensity (failures per thousand CPU hours) rather than the developer-oriented fault density (faults per thousand source lines) are both better in meeting the customer's needs and *much easier* to achieve.

Improving failure intensity is easier because you can take advantage of an operational profile that is usually nonuniform and concentrate your quality-improvement efforts on the functions used most frequently.

A simple example illustrates this point. For the same amount of testing, the failure intensity is proportional to the number of faults introduced into the code during development.[1] Let the proportionality factor be k. Assume that a system performs only two functions: function A 90 percent of the time and function B 10 percent. Suppose that the software contains 100 faults, with 50 associated with each function.

Improving the fault density by 10 times means that you must improve development over the entire program so that 90 fewer faults are introduced.

The current failure intensity is $0.9(50)k + 0.1(50)k$, or $50k$. If you concentrate your development efforts on the most frequently used function so it has no faults, the new failure intensity will be $0.1(50)k$, or $5k$.

You have reduced the failure intensity by a factor of 10 through improvements in development involving only a factor-of-two reduction in the number of faults (50 faults). A similar but more extended analysis could also take account of failure severity.

Reliability models. Models play an important role in relating reliability to the factors that affect it. To both represent the failure process accurately and have maximum usefulness, reliability models need two components:

- The *execution-time* component relates failures to execution time, properties of the software being developed, properties of the development environment, and properties of the operating environment. Failure intensity can decrease, remain fixed, or increase with execution time. The last behavior is not of practical interest. Decreasing failure intensity, shown in Figure 2, is common during system test because the removal of faults reduces the rate at which failures occur. Constant failure intensity usually occurs for systems released to the field, when systems are ordinarily stable and no fault removal occurs.
- The *calendar-time* component relates the passage of calendar time to execution time. During test, it is based on the fact that resources like testers, debuggers, and computer time are limited and that these resources, each at a different time, control the ratio between calendar time and execution time. During field operation, the relationship is simpler and depends only on how the computer is used.

Life cycle

You apply software-reliability engineering in each phase of the life cycle: definition, design and implementation, validation, and operation and maintenance.[6] The definition phase focuses on developing a requirements specification for a product that can profitably meet some set of needs for some group of customers. System and software engineers develop product designs from the product requirements, and software engineers implement them in code in the design and implementation phase. Test teams, usually independent of the design and implementation teams, operate the product in the validation phase to see if it meets the requirements. In the operation and maintenance phase, the product is delivered to and used by the customer. The maintenance staff responds to customer requests for new features and to reported problems by developing and delivering software changes.

Definition phase. Good product definition is essential for success in the marketplace. The primary output of the definition phase is a product-requirements specification. The product's failure-intensity objective should be explicitly included.

The first step in setting the failure-intensity objective is to work with the customer to define what a failure is from the customer's perspective. Next, you categorize failures by *severity*, or the effect they have on the customer. Determine the customer's tolerance to failures of different severities and willingness to pay for reduced failure intensities in each failure-severity category.

Looking at the customer's experiences with past and existing products will help both of you determine the value of reliability. A larger market reduces the per-unit cost of reliability, making higher reliability more feasible.

Another consideration is assessing the reliability of competitors' products. You must also determine the effects of meeting the failure-intensity objective on delivery date.

You can then use the information developed in each of these steps to establish failure-intensity objectives, trading off product reliability, cost, and delivery date.

Customers generally view the foregoing communication very favorably. It greatly improves the match between product characteristics and customer needs, and it generally increases the customer's trust in the supplier.

Now, you need two other items:

- The operational profile, since the product's reliability may depend on how the product will be used.
- Estimates relating calendar time to execution time, so that failure-intensity objectives expressed in terms of calendar time (the form customers can relate to) can be translated into failure-intensity objectives expressed in terms of execution time (the form relevant to software).

So you can readily determine failure intensity in testing and in the field, you should consider building automatic failure identification and reporting into the system.

Two questions often asked about applying models during the definition phase are:

- How accurate are the reliability predictions made at this time?
- Is the effort to model reliability at this point worth it?

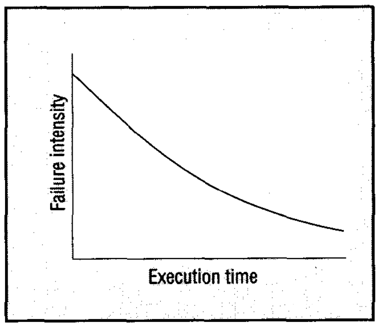


Figure 2. General appearance of the software-reliability model's execution-time component.

There is not enough information to give a definitive answer to the first question today, although there are some indications that the accuracy may be within a factor of three or four. Accuracy is likely to be determined and to improve as the field progresses and appropriate data is collected.

As to the second question, the effort would probably not be worth it if the modeling were carried no further than just predicting software reliability. However, the modeling should continue into the design and implementation, validation, and operation and maintenance phases. Particularly during the validation and the operation and maintenance phases, you can use modeling with collected data to track whether reliability objectives are being met. Also, you can use the results to refine the prediction process for the future.

The real utility of modeling becomes evident in this cycle of model, measure, and refine. Furthermore, applying models during the definition phase forces project teams to focus very early on reliability issues and baseline assumptions about the product's reliability *before* development begins.

Design and implementation phase. The first goal of the design and implementation phase is to turn the requirements specification into design specifications for the product and the development pro-

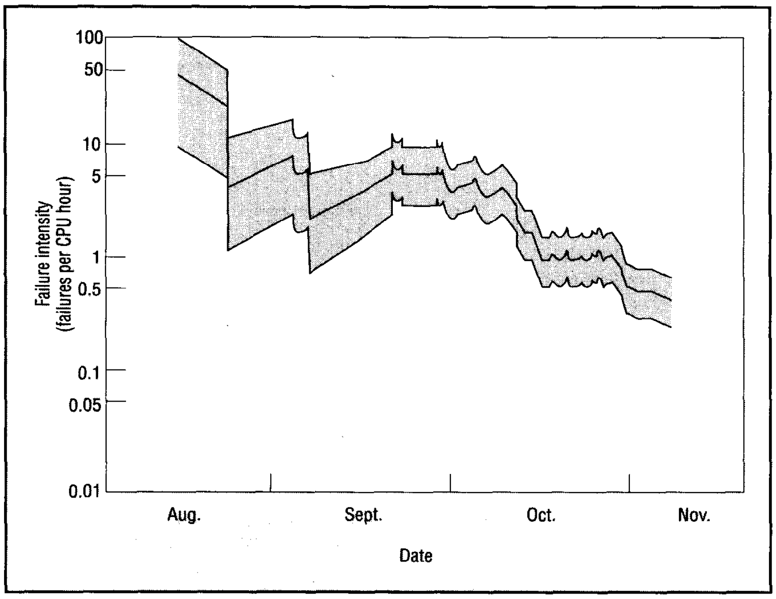


Figure 3. Tracking reliability status in system test. The center curve indicates the most likely value of failure intensity. The shaded area represents the interval within which the true failure intensity lies with 75-percent confidence. The *y* axis is logarithmic so the data can fit the graph.

cess. Then the process is implemented.

An early activity of this phase is allocating the system-reliability objective among the components. In some cases, you may want to consider different architectural options. You should analyze whether you can attain the reliability objective with the proposed design.

It will usually be necessary to identify critical functions for which failure may cause catastrophic effects. You should then identify the modules whose satisfactory operation is essential to the functions, using techniques like failure-modes and -effects analysis and fault-tree analysis. You can then single out the critical modules for special fault-avoidance or fault-removal activities. You may also use fault-tolerance techniques like periodic auditing of key variables during program execution and recovery blocks.

The design of the development process consists of determining the development activities to be performed with their associated time and resource requirements. Usually, more than one plan can meet the product requirements, but some plans are faster, less expensive, or more reliable than others.

The software-reliability-engineering part of the development-process design involves examining the controllable and uncontrollable factors that influence reliability. Controllable factors include such things as use or nonuse of design inspections, thoroughness of design inspections, and time and resources devoted to system test. Uncontrollable (or perhaps minimally controllable) factors are usually related to the product or the work environment; for example, program size, volatility of requirements, and average experience of staff.

You use the uncontrollable factors to predict the failure intensity that would occur without any attempt to influence it. You then choose suitable values of the controllable factors to achieve the failure-intensity objective desired within acceptable cost and schedule limits. Techniques now exist to predict the effects of some of the factors, and appropriate studies should be able to determine the effects of the others. The relationships between the factors and reliability must be expressed in simple terms that software engineers can intuitively understand. Otherwise, they are not likely to apply them.

You should construct a reliability time line to indicate goals for how reliability should improve as you progress through the life cycle from design through inspection to coding to unit test to system test to release. You use this time line to evaluate progress. If progress is not satisfactory, several actions are possible:

- Reallocate project resources (for example, from test team to debugging team or from low-usage to high-usage functions).
- Redesign the development process (for example, lengthening the system-test phase).
- Redesign subsystems that have low reliability.
- Respecify the requirements in negotiation with the customers (they might accept lower reliability for on-time delivery).

Reliability can't be directly measured in the design and coding stages. However, there is a good chance that indirect methods will be developed based on trends in finding design errors and later trends in discovering code faults by inspection or self-checking.

Syntactic faults are usually found by compilers or perhaps editors, so you should concentrate on *semantic* faults to predict reliability. In unit test, we anticipate that methods will be developed to estimate reliability from failure trends.

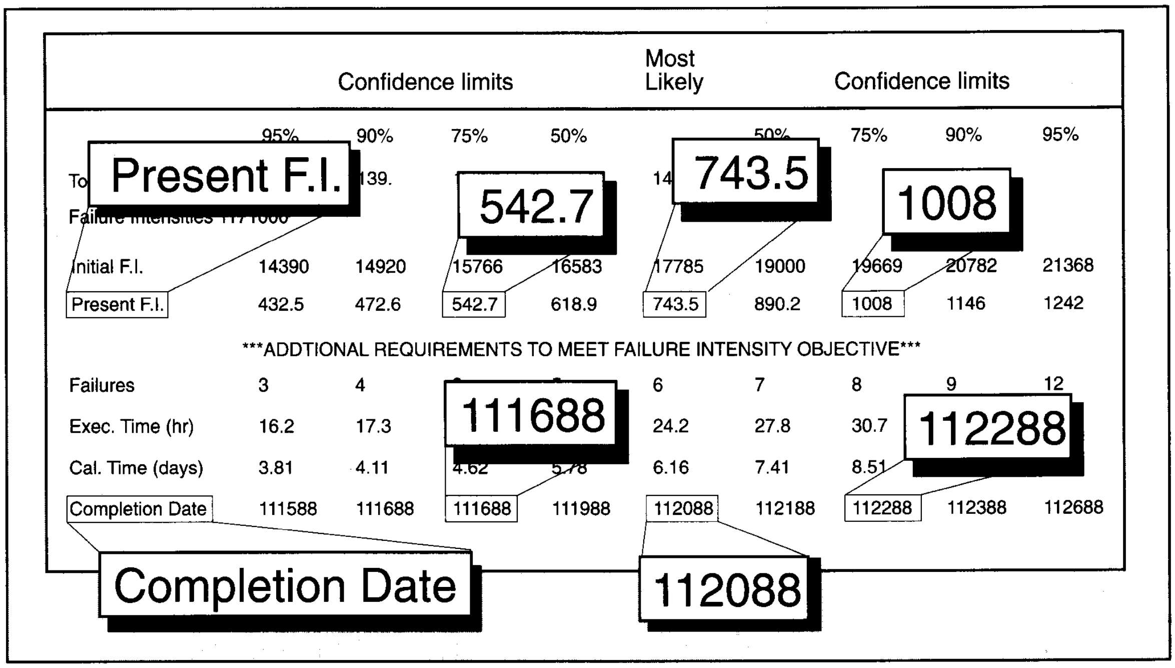
Another important activity is to certify the reliability of both acquired software and reused software (not only application software but also system software like operating-system and communication-interface software). You should establish the reliability of such components through testing with the operational profile expected for the new product.

The operational profile can help increase productivity and reduce cost during the design and implementation phase by helping guide where you should focus design resources.

Verification activities like inspections, unit test, and subsystem test are commonly conducted during the design and implementation phase. You can apply inspections to both design and code.

Validation phase. The primary thrust of validation is to certify that the product meets customer requirements and is suitable for customer use. Product validation for software generally includes system tests and field trials.

Software-reliability measurements are particularly useful in combination with reliability testing (also called longevity or stability testing). During reliability testing, you generally execute functions with relative frequencies that match what is specified in the operational profile. When fail-


Present F.I.
542.7
743.5
1008
111688
112288
Completion Date
112088

	Confidence limits				Most Likely	Confidence limits			
	95%	90%	75%	50%		50%	75%	90%	95%
To[illegible]	[illegible]	139.	[illegible]	[illegible]	14[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Failure intensities [illegible]									
Initial F.I.	14390	14920	15766	16583	17785	19000	19669	20782	21368
Present F.I.	432.5	472.6	542.7	618.9	743.5	890.2	1008	1146	1242
ADDTIONAL REQUIREMENTS TO MEET FAILURE INTENSITY OBJECTIVE									
Failures	3	4	[illegible]	[illegible]	6	7	8	9	12
Exec. Time (hr)	16.2	17.3	[illegible]	[illegible]	24.2	27.8	30.7	[illegible]	[illegible]
Cal. Time (days)	3.81	4.11	4.62	5.78	6.16	7.41	8.51	[illegible]	[illegible]
Completion Date	111588	111688	111688	111988	112088	112188	112288	112388	112688

Figure 4. Sample printout from Reltools. The present failure intensity lets you determine whether the failure-intensity objective has been met. Key quantities are called out here for emphasis.

ures are experienced, developers start a correction process to identify and remove the faults causing them.

You may use multiple operational profiles, each relating to a different application with its associated market segment.

We expect it will be possible in the future to test using strategies with function-occurrence frequencies that depart from the operational profile. You would compensate for this departure from the profile by adjusting the measured failure intensities.

To obtain the desired reliability quantities, you first record failures and the corresponding execution times (from the start of test). Then you run a software-reliability-estimation program like Reltools (available from the authors), which uses statistical techniques to estimate the parameters of the software-reliability model's execution-time component, based on the recorded failure data. Applying the execution-time component with the estimated parameters, it determines such useful quantities as current failure intensity and the remaining execution time needed to meet the failure-intensity objective. It also uses the calendar-time component to estimate remaining calendar time needed for testing to achieve the failure-intensity objective.[7]

Managers and engineers can track reliability status during system test, as Figure 3 shows. The plot shows real project data. The downward trend is clear despite real-world perturbations due to statistical noise and departures of the project from underlying model assumptions. The actual improvement in failure intensity is about 70 to 1, a fact deemphasized by the need to use a logarithmic scale to fit the curves on the chart.

Problems are highlighted when the downward trend does not progress as expected, which alerts you before the problem gets too severe. Cray Research is one company that applies reliability tracking for this purpose as standard practice in its compiler development.

Figure 4 shows a sample printout produced by Reltools. The way it presents current failure intensity lets you determine whether the failure-intensity objective has been met. You can use this as a criterion for release, so you are confident that customer needs are met before shipping. There have been several demonstrations of the value of software reliability as a release criterion, including applications at AT&T[8] and Hewlett-Packard.[9] Cray Research[10] uses such a release criterion as standard practice.

Operation and maintenance phase. The primary thrust of the operation and maintenance phase is to move the product into the customers' day-to-day operations, support customers in their use of the product, develop new features needed by customers, and fix faults in the software that affect the customers' use of the product.

For those organizations that have operational responsibility for software products, reliability measurement can help monitor the operating software's reliability. You can use the results to determine if there is any degradation in reliability over time (for example, degradation caused by the introduction of additional faults when software fixes are installed). Software-reliability measurement can help you time the addition of new software features so reliability is not reduced below a tolerable level for the user.

Field-support engineers can use operational software-reliability measures to compare the customer's perceived level of reliability to the reliability measured by the supplier at release. Several factors could cause these measures to differ: different definitions of "failure," different operational profiles, or different versions of the software. Determining which factors contribute to the differences and feeding information back to the appropriate people is an important task.

You can also apply software-reliability engineering to maintenance. One prime example is using the frequency and severity of failures to rank the order for repairing underlying faults. In addition to other considerations, users usually consider failures in old functions more severe than those in new functions because their operations are likely to be more heavily affected by the old functions they depend on.

You can also use software-reliability-engineering methods to size the maintenance staff needed to repair faults reported from field sites. You can use the methods to estimate PROM production rates for firmware. Also, you can use them to estimate warranty costs. A potential use is to determine the optimum mix of fault-removal and new-feature activities in releases.

Process improvement. The most important activity that you can conduct near the end of the life cycle is a *root-cause analysis* of faults. Such an analysis determines when and how the faults are introduced and what changes should be made to the development process to reduce the number introduced in the future.

Also, if you have used a new technique, tool, or other "improvement" to the software-engineering process, you should try to evaluate its effect on reliability. This implies a comparison with another situation in which all product and process variables are held constant except the one whose effect is being checked. That can be difficult across projects, so it may be more feasible to test the "improvement" across subsystems of the same project.

Research opportunities

The applications of software-reliability engineering described here have in most cases been based on existing capabilities, which in some instances have already been used on a pilot basis. However, some depend on extensions requiring studies, usually based on actual project data. The structure of this research appears reasonably clear, and in many cases it is well under way. There is of course some risk that all the extensions might not come to fruition.

Predicting reliability. The largest and most significant area of potential advance is the prediction of reliability before program execution from characteristics of the software product and the development process. Initial work[1] indicates that failure intensity is related to the number of faults remaining in the software at the start of system test, the size of the program in object instructions, the average processing speed of the computer running the software, and the fault-exposure ratio. The fault-exposure ratio represents the proportion of time that a hypothetical encounter of a fault, based on processing speed and program size, would result in failure.

Some evidence indicates that the fault-exposure ratio may be a constant. This needs to be verified over a range of projects. If not constant, it may vary with a few factors like some measures of program "branchiness" and "loopiness." These relationships would need to be developed.

The number of faults remaining at the start of system test is clearly related to program size. You can approximate it by the number of faults detected in system test and operation, provided the operating period totaled over all installations is large and the rate of detection of faults is approaching zero.

Studies indicate that it depends (to a lesser degree) on the number of specification changes, thoroughness of design documentation, average programmer skill level, percent of reviews accomplished, and percent of code read. It is possible that adding more factors would further improve such predictions; finding out requires a study of multiple projects.

The selection of appropriate factors might be aided by insight gained from root-cause analysis of why faults get introduced. Our experience indicates that the factors are most likely related to characteristics of the development process rather than the product itself.

Interestingly, complexity other than that due to size seems to average out for programs above the module level. Thus, it is not an operative factor. This fact appears to support the conjecture that the fault-exposure ratio may be constant.

Estimating before test. An area of challenge is to find a way to estimate the number of remaining faults based on patterns of data taken on design errors detected in design inspections and coding faults detected in desk-checking or code walkthroughs.

This estimate of remaining faults is needed to estimate software reliability before test for comparison with a reliability time line. One possibility would be to apply an analog of software-reliability-modeling and -estimation procedures to the values of inspection or walkthrough execution times at which you experience the design errors or code faults.

Estimating during test. Work is needed to develop ways of applying software-reliability theory to estimating failure intensity during unit test. There are two problems to deal with:

- the small sample sizes of failures and
- efficiently determining the operational profiles for units from the system's operational profile.

Estimating reliability from failure data in system test works fairly well, but improvements could be made in reducing estimation bias for both parameters and predicted quantities. Adaptive prediction, which feeds back the prediction results to improve the model, is showing considerable promise.

Other opportunities. Current work on developing algorithms to compensate for testing with run profiles that are different from the operational profile may open up a substantially expanded range of application for sofwtare-reliability engineering.

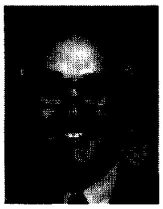
As software-reliability engineering is used on more and more projects, special problems may turn up, as is the case with any new technology. These offer excellent opportunities for those who can closely couple research and practice and communicate their results.

Software-reliability engineering, although not yet completely developed, is a discipline that is advancing at a rapid pace. Its benefits are clear. It is being practically applied in industry. And research is proceeding to answer many of the problems that have been raised. It is likely to advance further in the near future. This will put the practice of software engineering on a more quantitative basis. ❖

Acknowledgments
The authors are grateful to Frank Ackerman, Jack Adams, Bob Brownlie, Mary Donnelly, Anthony Iannino, Tom Riedl, Sheldon Robinson, and the other authors in this special issue for their helpful reviews.

References

1. J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.
2. J.D. Musa, "Tools for Measuring Software Reliability," *IEEE Spectrum*, Feb. 1989, pp. 39-42.
3. L. Bernstein and C.M. Yuhas, "Taking the Right Measure of System Performance," *Computerworld*, July 30, 1984, pp. ID-1–ID-4.
4. R.D. Buzzell and B.T. Gale, *The PIMS Principles: Linking Strategy to Performance*, The Free Press, New York, 1987, p. 109.
5. W.K. Ehrlich, J.P. Stampfel, and J.R. Wu, "Application of Software-Reliability Modeling to Product Quality and Test Process," *Proc. 12th Int'l Conf. Software Eng.*, CS Press, Los Alamitos, Calif., pp. 108-116.
6. W.W. Everett, "Software-Reliability Measurement," *IEEE J. Selected Areas in Comm.*, Feb. 1990, pp. 247-252.
7. J.D. Musa and A.F. Ackerman, "Quantifying Software Validation: When to Stop Testing?" *IEEE Software*, May 1989, pp. 19-27.
8. P. Harrington, "Applying Customer-Oriented Quality Metrics," IEEE Software, Nov. 1989, pp. 71, 74.
9. H.D. Drake and D.E. Wolting, "Reliability Theory Applied to Software Testing," *Hewlett-Packard J.*, April 1987, pp. 35-39.
10. K.C. Zinnel, "Using Software-Reliability Growth Models to Guide Release Decisions," *Proc. IEEE Subcommittee Software-Reliability Eng.*, 1990, available from the authors.

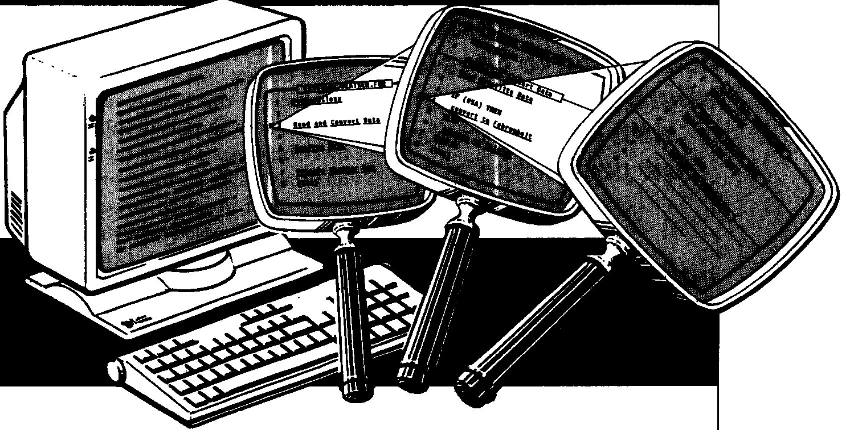


John D. Musa is supervisor of the Software Quality Group at AT&T Bell Laboratories. He has managed or participated in many software projects.
Musa received an MS in electrical engineering from Dartmouth College. He is a senior editor of the Software Engineering Institute book series, a contributing editor of *IEEE Software*, and an editor of *Technique et Science Informatiques*. He was elected an IEEE fellow for his extensive contributions to the fields of software engineering and software-reliability engineering over the past 15 years.

William W. Everett is a distinguished member of the technical staff at AT&T Bell Laboratories. His research interests include software engineering, software reliability, and computer-systems performance analysis.
Everett received an engineer's degree from the Colorado School of Mines and a PhD in applied mathematics from the California Institute of Technology. Everett is a member of the Society of Industrial and Applied Mathematics.

Address questions about this article to the authors at AT&T Bell Labs, Rm. 6E111B, Whippany Rd., Whippany, NJ 07981; Internet jdm@ whutt.att.com.

The key to effective maintenance is program comprehension.



SEELA Reverse Engineering

simply powerful

Using on-line reverse engineering in the VAX/VMS environment, SEELA displays your code top-down as a readable program-design language and lets you browse and edit the code through the PDL display, organize and simplify the code by program blocks, and generate high-level documentation describing the code structure.

NEED WE SAY MORE?
Well, maybe... SEELA has a built-in pretty printer, it supports all types of printers and terminals, it interfaces to CASE Design tools, it is simple to use and easy to introduce (it uses standard VMS editors), it works with many languages: Ada, C, Cobol, Fortran, PL/M, Pascal, etc. And, SEELA is very cost-effective!

Tuval

For more information and a free PC demo (specify language):
TUVAL Software Industries Ltd.
520 South El Camino Real, Suite 700, San Mateo, CA 94402-1720
Tel: 1-800-777-9996, Ext. 438, Fax: 001-415-342-9560

Visit our booth at Tri-Ada '90 and CSM-90