

Measuring software reliability

Code must be reliable from the surprisingly divergent viewpoints of software developers, testers, and users

Software reliability can mean different things to different people in different situations. A software developer, who views code as instructions to hardware, may judge them reliable if each software requirement is executed properly by a related set of those instructions. A user sees the same software as a set of functions and deems it reliable if nothing malfunctions.

The two viewpoints may yield contradictory estimates of the same software's reliability. For example, someone placing a telephone call at a time when all circuits are busy may be dropped mid-call by a system seeking to balance the load on the overall system and keep as many people as possible on the line. To the caller, the system looks unreliable; to the developer, it is working exactly as required.

Conversely, the same person calling from New York City to San Francisco is happy to find that calls always go through. But because of a fault in the system, all calls between those two points are routed through Chicago. The user thinks the system is reliable, but the phone company staff who maintain the system see the unusually high volume of traffic through Chicago and know something is wrong.

Examining the component concepts of that vague term "reliability" should clarify why its measurement is difficult. It should also reveal what issues require further exploration and definition before reliability measurement becomes as straightforward for software as for hardware.

MANY PRODUCTS. The notion of software reliability borrows heavily from its counterpart in hardware: a reliable system is one that works correctly over a long period of time. But in hardware, the final product—say, an automobile—is not the only product of development. There are many intermediate products: the requirements that specify

the automobile's characteristics and functions, the architectural layout of its parts and wiring, and the prototype built to test the reliability of the design before it enters manufacture. The final car will not be reliable unless the requirements, the design, and the individual parts are all reliable.

Similarly, computer code—1s and 0s—is only the final product of software development. The intermediate products are the requirements specifying the functions the software must perform; the design drawn from the requirements and used by the programmers to generate code; and even the plans for testing the software.

All those intermediate products of development can and should be considered when evaluating a software system's reliability. The requirements specify how reliable the code must be. The design must implement the requirements in such a way that the specified reliability is likely to be achieved. The test plans measure how well the code meets the requirements.

FINDING THE SOURCE. But testing may uncover not just mistakes in the code but also problems with the design and the requirements. Since correcting a problem in the requirements costs only 1 percent as much as fixing code, early reliability prediction can save money and time.

Fixing a problem in the requirements costs 1% as much as fixing the resulting code

To describe problems in software, the IEEE/American National Standards Institute (ANSI) Standard 982.2 distinguishes among errors, faults, defects, and failures. While all the definitions are related, differentiating among them assists software developers in pinpointing the source of a problem.

According to the IEEE/ANSI standard, an *error* is a human mistake that results in incorrect software. For example, a user may have omitted a critical requirement in the software specification, so that the software developers design an inappropriate product. Similarly, the developers may have misinterpreted a requirement, or translated incorrectly from design to code.

The resulting *fault* is an accidental condition that causes a unit of the system to fail

to function as required. Thus, the fault is a manifestation of an error in the software. Sometimes, faults are called bugs—parts of the software needing to be fixed.

Faults often lead to a *defect*—an anomaly in a product. Defects include problems not only with the code (the final product) but also with the design and requirements (intermediate products). Defects include an ambiguous requirement, an omission in a design document, a fault in code mature enough for test or operation, an incorrectly specified set of test data, an incorrect entry in the user documentation, and more. And it is defects that lead to failures.

A *failure* occurs when a functional unit of the software-related system (including the software-driven hardware) either can no longer perform its required function, or cannot perform it within specified limits.

CAUSE AND EFFECT. Human errors, software faults, and product defects describe the causes of problems, whereas functional failures describe the effects.

The cause of each problem must be traced to its root, because that determines the problem's impact on system reliability. A design flaw is often more serious than a simple software fault. Consider an analogous situation in automobile production, where testing has shown that a new car is likely to explode when hit from behind. If the root cause is that the gasoline tank is made from inferior metal, then the solution may simply be to find a new metal supplier. However, if the root cause is faulty design—the placement of the gasoline tank—then the automobile may need total redesign.

In the same way, an intermediate product may be ultimately responsible for unreliable software. A human may misinterpret a critical requirement, leading to a bad design, resulting in bad code. Such a chain of events is often harder to remedy than a misplaced comma in code.

Note that faults, defects, and failures represent dissimilar points of view. Faults and defects are what the developer and maintainer see: they view the system from the inside out, tracking faults and defects to find the cause of problems. On the other hand, failures represent a user's view of the system: the concern is how the system functions (or fails to), regardless of cause.

An error can occur from either point of view. The user may err in specifying the requirements for a system or in using the system. Alternatively, the developer may code

Shari Lawrence Pfleeger Mitre Corp.

incorrectly or misinterpret a requirement. Either error can lead to faults and/or failures. **TOUGH TO MEASURE.** The distinctions among errors, faults, defects, and failures are important because reliability, often viewed in terms of software faults (the developer's view), is officially defined in terms of functional failures (the user's view).

As defined by IEEE/ANSI Standard 982.2, software reliability is "the probability that software will not cause the failure of a system for a specified time under specified conditions." This definition parallels that of the hardware world. But any attempt to measure software reliability quantitatively, according to the standard IEEE/ANSI definition, runs into several problems.

First, the standard IEEE/ANSI definition is not accepted by everyone who writes reliability requirements. The interpretations of the term as expressed in user requirements are many and varied. Some indeed follow the IEEE/ANSI definition and deal with the user's view of the system (looking at failure information). Others focus on characteristics of the code as seen by the developer (looking at faults). Different measures may be needed to capture each.

Second, the standard requires tracking the use of the system over time while noting the number of failures. But when reliability requirements are strict, it may be impossible to test the system for long enough to verify a very low probability of failure. Or the system may be untestable in the field, as are several of the systems proposed for the U.S. Strategic Defense Initiative.

Perhaps most serious, the standard IEEE/ANSI definition requires the system to be completely designed, developed, and operational before reliability can be measured. That leaves developers with no direct, preliminary measures of reliability while the software is being written—even though software is harder and more costly to fix when complete than while still in development.

REAL WORLD. Broadly speaking, there are two approaches to measuring the reliability of finished code. The first, a developer-based view, focuses on software faults; if the developer has grounds for believing that the

1. Some existing software reliability indicators

Measure	Purpose	Definition	When to use
Required reliability	Indicates requirements for reliability	Quantitative rating from very low to very high, based on Cocomo cost driver ^a	For the requirements specification
Run reliability	Predicts final software reliability	Given k randomly selected runs during a specified time, probability that all k runs give correct results	Whenever set of possible discrete input patterns and states is well-defined
Fault density	Predicts remaining faults	Number of faults divided by thousands of lines of code (reported by severity level ^b)	During testing
Mean time to discover next k faults	Predicts time to reach reliability goal	Based on certain reliability models, observed times until last failure divided by number of failures from start of test to now, plus observed time between the two failures	During testing
Independent process reliability	Measures service reliability	Sum of correctness delta function weighted by user profile ^c	During testing, when processes are still logically independent and can be tested separately
Failure rate	Indicates growth in reliability as a function of test time	Cumulative probability distributions based on certain reliability models	In acceptance testing
Mean time to failure	Predicts stability of system	Mean observed times to next failure (either clock time or execution time)	In acceptance testing

^a Cocomo stands for Constructive Cost Model, a cost-estimation technique.

^b Severity level is how much impact a failure may have on a system ranging from, for example, loss of income through loss of subsystem function or of total system to loss of life.

^c The correctness delta function maps a correct response as 1, an incorrect one as 0, weights the results by the likely use of each function, and adds them to yield a figure of merit.

system is relatively fault-free, then the system is assumed to be reliable. The second, a user-based view more in keeping with the standard IEEE/ANSI definition, emphasizes the functions of the system and how often they fail.

In the first approach, the developer may collect information about fault density: the number of those faults discovered per 1000 lines of code. The developer then tracks the total number of unique faults in a given time interval. This number of faults divided by the total number of lines of code in the final product yields the fault density. By comparing fault density numbers for similar systems, the developer can judge whether the current system has been tested thoroughly. In

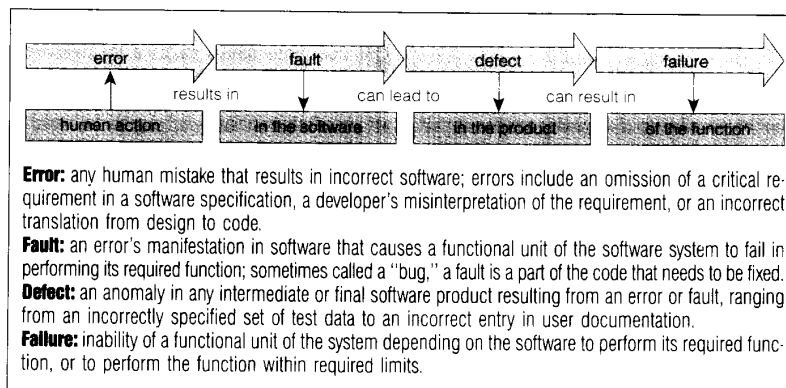
addition, the developer can infer the likely reliability of the software once it is installed in the field.

Alternatively, the developer may use a technique called fault seeding to estimate the number of faults remaining in the existing software. The quality-control team deliberately inserts into the software faults representative of the kinds observed to have occurred in the past (such as transposed 1s and 0s, or an incorrect exponent, or a branch to a wrong place in the code). The distribution of the faults matches the probability distribution previously observed on similar projects.

Then, the test team searches the code for all faults. In theory, if they discover all the seeded faults, the testing has been thorough enough to uncover all the accidental unseeded faults as well. The ratio of discovered non-seeded to seeded faults is taken to indicate the number of faults remaining in the code, the thoroughness of testing, and—indirectly—the system's reliability.

THE FAULT WITH FAULTS. The problem with the fault-seeding approach is that it does not look at failures in any context. Faults may exist in the code but not affect user function (for example, in income-tax-computation software, a fault may cause certain programmed equations not to work in the case of someone having negative income—but since no one has negative income, the fault does not matter). The system is thus faulty but reliable, according to the standard IEEE/ANSI definition.

Defining terms



Moreover, all the fault-based techniques can be misleading: the fact that many faults are discovered can mean that testing is thorough; but it can also mean that many more faults remain in the code.

For this reason, John Musa, supervisor of software quality at AT&T Bell Laboratories in Whippany, N.J., and others have concentrated on a failure-based approach to reliability. Musa broke new ground in 1975 by defining software reliability as how closely user requirements are met by a computer program in actual operation. His definition incorporates the profile of the user's probable usage of the various system functions (an "operational profile") and counts in only the time the system is running, rather than clock time. Musa suggested that reliability be considered as the probability of failure-free operation of a computer program for a specified time—for example, software reliability of 0.92 for 8 hours of execution.

One technique uses failure profiles, tracking failures in several categories of severity level. That is, failures can be classified in terms of the severity of their effect(s) on the system. For example, developers of a telecommunications system may use three levels of severity. Level 1 is minor, resulting at most in loss of income (example: system consistently underreports connection time by one minute when billing). Level 2 is major, producing partial loss of function (example: system cannot forward calls for all exchanges). Level 3 is critical, entailing total loss of a key function (example: no calls can be made to the East Coast). Cumulative failures may be tracked over time, where time may be measured in a variety of ways, including execution time, clock time, and usage time. The failure profile may be viewed for the overall system, for subsystems, and even for modules. The shape of the resulting curve is then used to project when testing will be complete, assuming sufficient test coverage.

Another technique is the analysis of time between failures. A model of failure rate is chosen (based on which models have been most accurate for similar software applications), and estimates are made of initial fault content (number of faults), normalizing constants (distribution of faults throughout the code), and initial mean time to failure. The model is then used to estimate the number of remaining faults, the ultimate mean time to failure, and hence the system's overall reliability.

Many such failure-rate models exist, including some based on mathematical models of a non-homogenous Poisson process and on a Bayesian model. As might be expected, choosing the correct model is a challenge with failure-based approaches.

All those approaches, however, depend on

2. Levels of reliability of code

Type of testing	What is tracked	Reliability measured			
		Modules	Subsystems	System	Functions
Unit	Faults	X			
Integration	Faults	X	X		
System	Faults	X	X	X	
Acceptance	Failures	X	X	X	X

the software system's being nearly complete and look primarily for mistakes in the code. Ideally, well before the system is cast into code, there should be some way of evaluating the reliability of all the early and intermediate products, including the requirements and the design, so as to predict the reliability of the code.

As of now, no proven ways exist of evaluating the reliability of a system's requirements or design. At best, the probability that the design will result in reliable code can be assessed only in terms of measurements that may indicate the likely reliability. Examples are design complexity, the number of defects discovered in a design review, or the density of defects relative to some measure of the design's size [Table 1].

Note that none of these indicators is a direct measurement of reliability, nor is any of them a foolproof, accurate predictor of the required reliability. Neither do developers always have methods to examine intermediate code products to reassure them that the final code will meet the system's reliability requirements. At best, intermediate indicators suggest trends.

Thus, although reliability can be expressed in quantitative terms when defining the system's requirements, there are only indirect ways of controlling this feature during the system's development. Furthermore, although it may be possible to meas-

ure the completed system's reliability, it may not be possible to change the system at that point if it is found to be unacceptable.

Ideally, there should be some way of evaluating the reliability of the requirements and design

ure the completed system's reliability, it may not be possible to change the system at that point if it is found to be unacceptable. **THEORY VS. PRACTICE.** This gap between theory and practice—between predictions of reliability from indirect indicators (such as the number of design defects) and actual measurements of reliability (such as the number of failures over time)—is quite wide and narrowing very slowly. Several circumstances contribute to this situation.

First, regardless of whether reliability is defined in terms of software faults or func-

tional failures, it is difficult to generalize from measurements of the reliability of software subsystems to the likely reliability of the system as a whole. There are a host of reasons, among them the fact that the reliability of the individual units indicates nothing about the reliability of the connections and sequencing among units. More fundamentally, Maria Teresa Mainini of Esacontrol, Genoa, Italy, and Luc Billot of CISI Ingénierie, Rungis, France, point out that testing various subsets of the system actually conveys different views of reliability and different test results [Table 2]. Unit testing, integration testing, and system testing, for example, which each focus on identifying software faults, say something about the reliability of individual modules, of subsystems (that is, collections of reliable modules) and of the system, respectively. But it is not until acceptance testing that the test team looks directly at functional failures to evaluate the system's actual operational reliability and to verify its functionality as specified in the requirements.

To be sure, in some cases the earlier measures (of faults) can be linked to predict what later measures (of failures) are likely to be. But as pointed out earlier, faults in the code are not the only causes of failure. For example, users may misuse the system because of a poor design. Thus, the indicators of software reliability should be viewed in the context of what question is being asked. Otherwise an inference about ultimate reliability from earlier indicators could be inappropriate.

Finally, even when there is agreement on a definition of reliability and it is measured in the correct context, a simple measure may not be enough to capture the measurer's intent. If so, it is better to view reliability as a composite of measures of the various products of development in the context of the reliability goals. In fact, in many cases, the reliability of the earlier products sets an upper limit to the reliability of the final product. For example, an inherently unreliable design may never yield reliable code.

A composite approach to reliability is timely. For years, software engineering researchers have defined simple one-dimensional metrics to measure complex things: usability, design complexity, maintainability, and more. Only now are researchers acknowledging that a broader perspective is needed. The simple measures must be combined to form a multidimensional composite, and for complete understanding, each measure must be interpreted in context.

THREE LEVELS OF MEASUREMENT. At least three steps are needed to implement this composite approach for reliability indicators at the early stages in software development, as well as help developers choose which set of mathematical models is appropriate in the

evaluation of software faults or functional failures at later stages. The steps involve defining reliability, identifying what about it needs to be known, and then measuring a variety of characteristics that help answer critical questions [Table 3].

Step one is crucial, since reliability presents a different aspect to developer, maintainer, and user. Each definition must be placed in the context of an overall system view of someone who understands the need for reliability.

The importance of this top level cannot be underestimated. Consider what happens when an error occurs in an airplane's software. Usually, the supporting system tries

to compensate so as to maintain system integrity. No general, systemwide failure occurs; the plane must keep flying no matter what, so each contributing subsystem must keep working.





In a telephone system, though, a similar error is usually handled by restarting an entire system or subsystem: the problem call is dropped, and the remaining system is intact. Such an approach is appropriate for communications but is out of the question for aerospace applications. Each system's needs dictate opposite interpretations of reliability.

Next, at the middle level of the reliability measurement framework, the overall defi-

nition of reliability is used to develop a model of the system from a given viewpoint. The model should show how and when the viewer must predict or verify the current reliability of the system. To illustrate, a system model can show how reliability is assessed from the design, is predicted from design and code characteristics, and improves during testing. This model makes it possible to determine the reliability goals of the system and to pose questions about those goals that measurement can answer.

To revert to the phone system and airplane: the model for the first reflects the definition that a failure may be counted only when the entire system ceases to work,

3. Three levels of reliability measurement

Defining what is meant by reliability				
Beholder	View	Model	Definition	
User	Pilot	Airplane is a collection of interdependent key subsystems	Total system works properly	
	Telephone caller	Network is a collection of independent but connected subsystems	A chain of subsystems works properly	
Developer	Airplane control software developer	Software controls a collection of interdependent key subsystems	All key subsystems work properly	
	Switch software developer	Software connects available independent subsystems	At least one chain of key subsystems works properly	
Maintainer	Airplane control software maintainer	Software enhancement at the least maintains system reliability by maintaining or increasing the reliability of each key subsystem	Same as for airplane control software developer	
	Switch software maintainer	Software enhancement at the least maintains system reliability by maintaining or increasing the number and reliability of connections	Same as for switch software developer	
Setting reliability goals				
Beholder	Reliability view	Reliability goal	Reliability questions	Reliability indicators/measures
Pilot	Sees airplane as the sum of its subsystems 	Maintain flight	Can plane maintain flight at all times?	Number of failures of total system over time Number of failures of key subsystems over time
Telephone caller	Sees telephone system as a connection between self and called party 	Obtain and maintain connection	Can caller be connected with called party? Can caller and called party stay connected for duration of call?	Number of disconnects (failures) over time
Airplane control software developer/maintainer	Sees airplane as a sequence of interdependent subsystems (sequential reliability) 	Maintain flight	Can overall system maintain flight? Can key subsystems meet required reliability?	Number of failures of total system of time Number of failures of key subsystems over time Number of faults discovered per subsystem Depending on goals, reliability of system can be the minimum, maximum, or product of subsystem reliabilities
Switch software developer/maintainer	Sees telephone system as a multitude of parallel connections (parallel reliabilities) 	Maintain maximum number of connections	Does any new connection lower system reliability? What is the minimum acceptable system reliability?	Number of failures (fewer than acceptable connections) over time Number of individual connections dropped over time Depending on goals, reliability of system can be the minimum, maximum, or product of 1 minus subsystem reliabilities
Measuring reliability (at any stage)				
Set initial goals for software reliability → Collect data on early software products (e.g., design defects) → Analyze data for indicators of reliability of final system → Predict reliability of actual system → Develop final code → Collect faults/failures data → Analyze data for actual reliability → Revise reliability goals, if necessary → Answer reliability questions → Correct code → Meet reliability goals				

How much testing is enough?

When is software deemed to be reliable enough to release to the customer? Software developers vary in how they combine software-reliability measures and models to make that decision. One decision technique is the zero-failure method of Motorola Inc., headquartered in Schaumburg, Ill., outlined by Ralph Brettschneider in the July 1989 issue of *IEEE Software*.

The zero-failure method specifies the number of hours of testing needed without a failure before the software is deemed ready for release to the customer. If even one failure is detected during this target test time, then the request for release to the customer is denied and testing must be continued.

According to Brettschneider, the zero-failure method—like other reliability models—is based on several key assumptions, of which two are particularly important.

First, the longer that testing proceeds without a failure, the more likely it is that the number of failures remaining is very small. Specifically, the zero-failure method assumes that the rate at which failures are discovered decreases exponentially as testing progresses.

Second, the zero-failure method assumes that testing is representative of the actual use of the software in the field, and that the probability of discovering failures is constant and equal for all kinds of failures.

To apply the method, Motorola sets a reliability goal in terms of an average number of failures per thousand lines of code. To calculate the number of failure-free test hours required, three inputs are needed:

ed: the permissible average number of failures due to faults embedded in the code sent to the customer; the total number of test failures detected so far; and the total number of hours the software has been tested up to the discovery of the last failure.

The Motorola zero-failure method sets forth a formula for calculating the number of hours needed to test without failure before the software can be deemed reliable.

As an example, Brettschneider considered a 33 000-line revision to a program. Up to now, 500 hours of execution time have revealed 15 repairable failures. No failures have been discovered in the past 50 hours since the last repair. If the goal is to deliver a product with no more than one failure (1 out of 33 000 lines or 0.03 failure per thousand lines of code), has the software now been tested enough?

Using the formula, Brettschneider showed that Motorola's test organization must test the 33 000 lines of code for another 27 hours with no failure discovered before the software is deemed ready to ship to the customer. If a failure occurs, however, the clock must be reset and testing must continue until there are 77 continuous hours without a failure.

Brettschneider's example shows that while measurement cannot ensure reliability, it can guide the development process and minimize the probability of unreliable software. Just as Motorola has done, other software developers can build or borrow a reliability model based on past experience and use it to increase confidence in the effectiveness of their testing.

—S.L.P.

whereas the model for the airplane deems it a failure when any key function stops working correctly.

Finally, the bottom level addresses the actual measurement values. During data collection and analysis, where information is gathered to instantiate the model, questions are answered, the answers viewed in the context of the overall system, and decisions made accordingly. At this lowest level, the measures of reliability may support quite contrary decisions for seemingly comparable failures. In the extreme, a telephone system with 100 dropped calls (partial failures) and zero complete system failures may be considered reliable, while an airplane with the same track record may be a dead loss.

Approaching reliability in this way provides a three-level framework for the process of reliability measurement: build a process or system model to depict the relevant issues, decide what questions are to be answered and define measures that address the questions, and gather information to help find the answers.

BURNING ISSUES. Models and measures of hardware reliability have been invaluable in assuring users that their hardware will function properly over time. So it is encouraging that the same notions are being applied to software reliability in the hope of generating similar assurances—especially when software has become a key part of almost

all systems that are in use today.

But current efforts have a long way to go. Close to home, the telephone system is still subject to embarrassing failures due to unreliable software: a widespread telephone outage in June and July 1991 was caused by one line of a subcontractor's code that had not been tested thoroughly [*"Faults & failures," Spectrum*, May 1992, p. 52].

Clearly, several issues must be addressed, or addressed more completely, before complete confidence can be reposed in the ability to predict and measure software reliability. At a minimum, researchers must:

- Build a family of traditional and non-traditional models of reliability, including parameters to determine which model is best for a given situation.
- Generate indicators of reliability early in the development process.
- Define the reliability of artifacts other than code.
- Determine composite measures of reliability that reflect the reliabilities of related artifacts.
- Suggest additional techniques for using reliability information to guide software development and maintenance.

This blueprint for the future suggests areas for researchers to investigate.

TO PROBE FURTHER. The 1988 *IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software*,

IEEE Standard 982.2, discusses almost three dozen commonly used indicators of reliability, summarizing the underlying theory and giving examples of how the indicators can be used.

Bev Littlewood, in his article "Theories of Software Reliability: How Good Are They and How Can They Be Improved?," discusses the pros and cons of several popular models of software reliability. The paper was published in *IEEE Transactions on Software Engineering*, Vol. SE-6, no. 5, September 1980, pp. 489–500.

Ralph Brettschneider, in "Is Your Software Ready for Release?," published in the July 1989 issue of *IEEE Software*, pp. 100, 102, and 108, gives a concrete, mathematical example of how a tailored reliability model is used to decide when testing is complete at Motorola Inc.

In the January 1990 issue of *Software Engineering Journal*, Vol. 5, no. 1, pp. 27–32, Maria Teresa Mainini and Luc Billot in their paper "PERFIDE: An Environment for Evaluation and Monitoring of Software Reliability Metrics During the Test Phase" supply an example of the actual tools and techniques used to evaluate reliability on a large project.

John D. Musa's seminal paper on defining reliability as a measure of how closely user requirements are met is "A Theory of Software Reliability and its Application," published in the *IEEE Transactions on Software Engineering*, September 1975, Vol. SE-1, no. 3, pp. 312–27.

Musa, Anthony Iannino, and Kazuhira Okumoto, in their textbook *Software Reliability: Measurement, Prediction, Application* (McGraw-Hill, New York, 1987), survey numerous leading approaches to the subject, including the development of an operational profile, choice of a reliability model, and use of the profile and model to guide software testing.

Musa summarized some of his approaches in "Tools for measuring software reliability," published in *IEEE Spectrum*, February 1989, pp. 39–42.

The second edition of *Software Engineering: The Production of Quality Software* (Macmillan, New York, 1991) by the author of this article discusses reliability in the context of the overall software development process.

ABOUT THE AUTHOR. Shari Lawrence Pfleeger (M) is a principal scientist at Mitre Corp.'s Software Engineering Center in McLean, Va., whose current research focuses on software metrics and the software development process. The author of two textbooks and more than two dozen journal articles in mathematics and computer science, Pfleeger's numerous IEEE activities include being a member of IEEE Software's Industrial Advisory Board. She also is a member of the Association for Computing Machinery, where she chairs the Committee on the Status of Women and Minorities in Computing. ♦