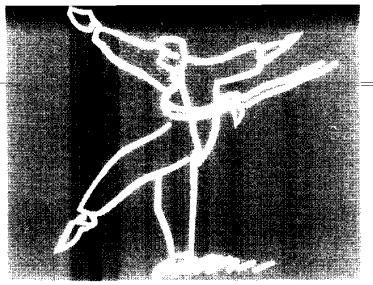


Reliability Measurement: From Theory to Practice

FREDERICK T. SHELDON,
General Dynamics, Fort Worth Division
KRISHNA M. KAVI, *University of Texas at Arlington*
ROBERT C. TAUSWORTHE,
Jet Propulsion Laboratory, Caltech
JAMES T. YU, *AT&T Bell Laboratories*
RALPH BRETTSCHNEIDER, *Motorola*
WILLIAM W. EVERETT, *AT&T Bell Laboratories*

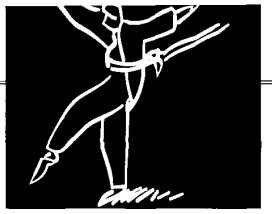
In today's climate of tight budgets and schedules, reliability measurement can help you deliver the level of reliability your customers need.

Pressure on software engineers to produce high-quality software and meet increasingly stringent schedules and budgets is growing. In response, reliability measurement has become a significant factor in quantitatively characterizing quality and determining when to release software on the basis of predetermined reliability objectives. In fact, some US Dept. of Defense organizations now require software reliability measurements.

Motorola's World Wide Systems Group has used reliability methods to help gauge the level of latent defects in hundreds of releases and millions of lines of code.[1] According to Ralph Brettschneider, Motorola has found reliability measurement to be exceedingly useful in helping decide if a release's quality is acceptable as measured against an ever improving quality goal: customer expectations. Brettschneider claims that reliability measurement for release control, plus other initiatives, has in the last four years produced a hundredfold improvement in the number of defects per thousand lines of source code reported on released software.

Despite similar testimonials, there is still a large gap between reliability measurement theory and practice. We seek to bridge this gap by presenting a few key issues that underlie reliability

0740-7459/92/0700/0013/$03.00 © IEEE



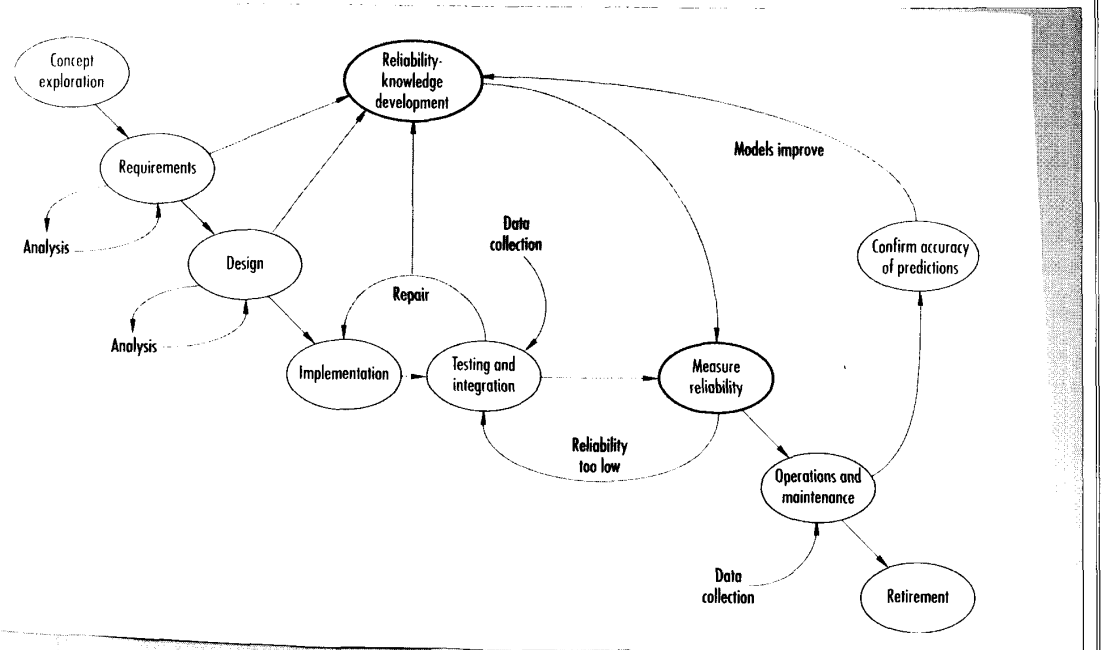


Figure 1. Reliability measurement happens in conjunction with testing and integration, before the software is released into operations and maintenance. Reliability-model development is fed by activities in the requirements, design, repair, and operations and maintenance phases. During reliability-model development, you plan how to use the selected model, set a reliability objective, and initiate activities to support the level of sensitivity you need for data collection (calendar time, wall-clock time, or CPU execution time, for example). Reliability data collected from fielded software can be useful for evaluating the accuracy of predictions and recalibrating the reliability model. The reliability model, which incorporates project-specific constraints, tolerances, and sensitivities, should retain this information so that it yields more accurate measures when it is reused on future projects.

measurement's evolution from theory to practice. We served on a panel at the Symposium on Applied Computing[2] in Fayetteville, Arkansas, where we outlined reliability measurement's salient issues, basic concepts, and underlying theory, which we present in this article. We also answered three basic questions about the technology; our responses are summarized in the box on p. 16-17.

We do not recommend a specific method and make no conclusions. However, we do consider reliability measurement to be an important *emerging* technology. As our comments indicate, we have an earnest concern that software is frequently unreliable, and we believe that reliability measurement can be a very effective, customer-oriented way to determine and deliver the appropriate level of quality.

Developers must identify useful reliability metrics as well as the program's operational profile.

RELIABILITY IN THE LIFE CYCLE

Reliability is one important measure that can help developers understand, manage, and control a development process constrained by time and cost.

The need for a quantitative understanding of software quality — and hence reliability — and the factors that affect it (like operational environment, testing methods, tools, schedules, and cost) has spurred much research on improving our insight into the development life cycle. Figure 1 shows where reliability measurement fits in. The effort to improve reliability measurement is fed by efforts in the requirements, design, testing and integration, and operations and maintenance phases.

Influencing factors. Many factors influence the target system's reliability: What are its hardware/software elements? Will it evolve in function and/or desired reliability? Do its parts run at different speeds? What are the customer's expectations?

Developers must determine what reliability metrics are useful and what the customer considers a failure (including even minor deficiencies and anomalies). Fault-tolerant systems require that the developer carefully distinguish anomalous internal states that can be tolerated from the failures that affect a customer's operations. Customers, too, may want to classify

failures by their severity (if they would affect safety, threaten life, cause a loss of income, or be expensive to repair, for example).

The developer must also identify an operational profile by gathering information on how previous versions were used, estimating the use of new features, and verifying the resulting estimated profile with the customer.[3] The operational profile can help plan test cases and data collection (possibly classified in terms of both developer-oriented and user-oriented characteristics) and methods to compensate for special conditions. Special conditions may include the effect of instrumentation on hard real-time scenarios or accounting for the fundamental differences among unit test, integration test, systems integration, and acceptance test.

Measurement process. Reliability modeling, then, has three broad stages.[4]

- *Assessment.* The developer makes some assumptions about the environmental conditions under which the software will run. This is an important step because it is often infeasible to re-create the operational environment exactly.
- *Model development.* The developer derives mathematical formulas to estimate (or predict) useful system parameters like failure intensity, number of failures in an interval, and the probability distribution of failure intervals. The developer estimates these parameters from real data using statistical techniques like maximum likelihood estimation, least squares estimation, and Bayesian methods.
- *Measurement and estimation.* The developer uses these parameters to predict behavior and help plan, maintain, and upgrade software. However, reliability measurement is typically distinguished from reliability prediction in that prediction is based on static metrics, such as size and complexity, and measurement (or estimation) is based on the dynamic execution behavior — the failure data collected during system test.

Recent work by Yashwant Malaiya, Nachimuthu Karunanithi, and Pardeep Verma[4] shows that some models work better in some cases. They have presented empirical results from five fault-count models to compare the predictive validity of each model according to three types of predictability measures (goodness-of-fit, next-step predictability, and variable-term predictability).

DEFECTS AND RELIABILITY

Reliability is the probability of failure-free operation for a specified time in a specified environment for an intended purpose. A "failure" happens when a defect causes the software to operate inappropriately — when operation deviates from system requirements.

Reliability is the probability of failure-free operation for a specified time in a specified environment for a specified purpose.

The four traditional ways to report failures are based on time: time of failure, time interval between failures, cumulative failures experienced up to a time, and failures experienced within a time interval.[3] A new metric uses time-independent measures.[2,5] This test-case-intensity metric counts the number of test cases applied per thousand lines of code to measure the amount of stress applied to the software during test. When combined with an estimate of complexity, this metric may be a powerful indicator of reliability prediction and validation.

Defect removal. Figure 2 illustrates the defect-removal process as a state-transition diagram. Although failure occurrence is random and the probability distribution varies with time, failure behavior is affected by three principal factors:

- the number of defects (λ-defects),
- the test strategy and operational profile, and
- defect detection, removal, and possible reintroduction (ρ-defects). Many reliability models allow for imperfect debugging,

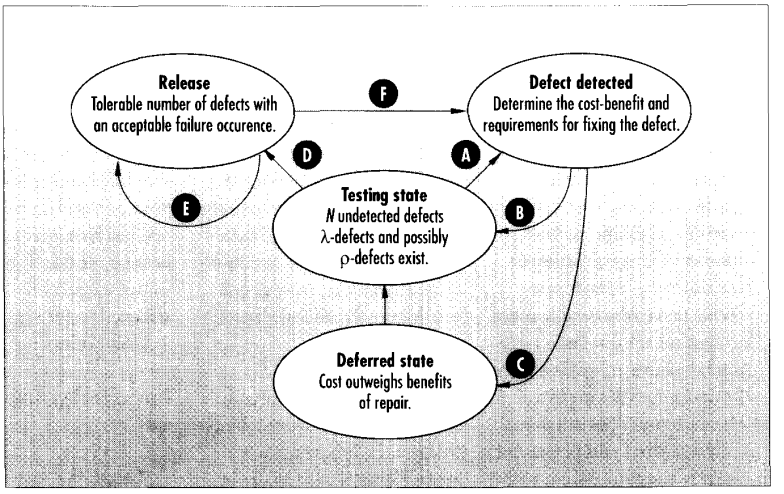


*Figure 2. The defect-removal process illustrated as a state-transition diagram. Beginning with the testing phase, a system with N unknown defects is executed until **(A)** a defect is detected. In the defect-detection phase, the developer can either **(B)** fix the defect and return the system to testing or **(C)** decide to defer repair for cost-benefit reasons. This process continues until **(D)** a sufficient amount of failure-free operation has occurred and the last intolerable defect has been repaired. Defects uncovered in the release state are considered either **(E)** tolerable or **(F)** intolerable, in which case the system is returned to the defect-detected state.*

MOVING FROM THEORY TOWARD PRACTICE: RELIABILITY PANEL COMMENTS

At the Symposium on Applied Computing, Frederick T. Sheldon and Krishna M. Kavi moderated a panel whose members have more than 35 years of experience in reliability measurement. Sheldon and Kavi asked each panelist to answer three basic questions about reliability-measurement technology. To conserve space, their responses are limited to different and unique comments.

Why do we need reliability technology?

Everett: There is a strong need for more customer-oriented measures of software quality. When I say this I am often asked, why do we need customer-oriented measures? Why not just build the best product we can? My answer is that without customer-oriented quality measures, we cannot consciously make effective trade-offs between cost and the delivery time frame. Unfortunately, trade-offs generally wind up being made unconsciously, with the level of quality being whatever fits into the cost and schedule constraints already established.

To the customer, one of the most important quality attributes is reliability. Current heavily used measures in software development, such as number of faults or faults per thousand lines of code, are closely tied to the development process but are not necessarily good measures of quality from the customer's perspective. The customer does not necessarily relate to faults per se, but rather to the failures they will cause. Failures, the frequency at which they occur, and their impact on business are measures more closely coupled to the customer's perception of quality.

Reliability tends to be used synonymously with hardware reliability. However, over the years the amount of software embedded in our products has increased to the point that, for many products, the reliability of the software dominates that of the hardware. The discipline of software-reliability engineering is emerging, and it will do for software what hardware-reliability engineering has done for hardware.

Tausworthe: There *is* a need for this technology. One main question concerns what the criteria are for a suitable model. It is necessary to distinguish between failure models and prediction models of the same phenomena. These need not be the same in all respects, but must be compatible in their assumptions about the underlying process. Outputs of prediction models should be related to risks, while outputs of failure models should be related to product and process data typical of that which can be measured.

There should be sound, intuitive, plausible, and verifiable assumptions basing the models. Formalism is necessary because empiricism is not enough. We have to understand the underlying physical process and then conform that process to empiricism. In this way, we may use empiricism to calibrate the models.

Yu: I would like to extend the scope of software reliability to cover the following two areas: Estimation of failure intensity — the number of software failures per unit of time — and prediction of remaining software faults.

Serious failures, such as outages, seldom occur in the testing environment, and software-reliability models have little use in such an environment. To apply software reliability, there must be enough software failures identified during the system testing interval. The operational profile constitutes the input data to test the software product. To simulate the customer environment, the frequency of each type of input data should also be collected from customers or similar products.

Brettschneider: Beyond the prime customer issue that software is too expensive lies a second major concern: Software is frequently unreliable. Though different, these two issues are related.

Failure to initially achieve reliable software will result in a need for additional testing and field support, the cost of which must be passed on to the customer. Unfortunately the release decision is usually based on an evaluation of the software's expected quality balanced against its release-date commitment. The cost of poor quality is then shifted from the producer to the consumer.

My experience in using a simplified decision-making approach, based on modeling theory, has been quite successful in helping me decide if acceptable software quality has been achieved and if the software is ready for release.

Is the technology ready for application?

Everett: We have seen tremendous growth in the science of software reliability in the last 15 to 20 years, in particular in the development of reliability models and numerical algorithms for evaluating such models. However, I feel the development of software reliability as an engineering discipline has lagged behind its development as a science.

More and more of the engineering discipline will evolve as we apply the science to our software.

My experience is that the basic theory is ready to apply now. As part of our education and training department at AT&T Bell Labs, I have been working with a number of projects to move software reliability from theory to practice. Some of the areas and ways in which software reliability has been and is being applied include

- monitoring the progress of system test,
- predicting elapsed system test time to achieve a specified reliability objective,
- defining operational profiles,
- setting reliability objectives,
- evaluating designs with respect to reliability,
- using reliability measures to change testing environments, and
- exploring how reliability measures can be used during development testing.

There tend to be enough initial benefits to justify the introduction of the technology, but we should keep its initial use simple. I have seen a number of successes in its application, in particular during system test, to monitor test progress and estimate test completion.

I would be remiss if I did not say we have also seen some setbacks. However, I have not seen any setbacks that I can relate to major deficiencies in the theory of software reliability.

There has been a lot of discussion over the controversy surrounding software reliability. Much of the controversy seems to center on which model is better, how well do the models reflect reality, and how well do models predict reality.

I think the controversy will continue around certain areas of the science of software reliability, and that's good. It will spur the further evolution of the science of software reliability so that it can stay ahead of the application of software reliability.

Some major challenges I have faced in applying the technology have not been from the lack of theory but from not understanding how to model how customers use software and how to set up appropriate test environments. By pushing for more application of software-reliability techniques within actual software development, we can speed the evolution of the engineering discipline of software reliability.

Tausworthe: Can the future reliability of software be predicted? Yes, undoubtedly — but it is a matter of accuracy and uncertainty as to whether the prediction is good enough. The uncertainty can be no better than the inherent random character of the underlying process. Use of parameter-estimation methods incorporating past-process data values can reduce the uncertainty, but only down to the inherent limit. The random deviation of the true physical process from sample function cannot be reduced in any model because it is independent of the model.

If the uncertainty of the underlying process is unacceptable and uncontrollable, it is useless to try to develop a prediction model to improve the uncertainty, because the effort will fail.

Yu: After estimating the model parameters, the user can perform extrapolation to predict failure intensity and software reliability. The user can also change the process (vary the model parameters) to achieve a higher quality level. Several applications of reliability analysis are

- establish criteria for determining when system testing is complete,
- predict what testing resources are required to achieve the quality objective,
- predict software quality at the time of product release, and
- predict the software faults remaining to plan support staff after product release.

Brettschneider: The only program outputs of real value are the number of latent defects left to be found and how much more testing is necessary to find them. The reliability goal should be to deliver a defect-free product.

Confidence intervals around software reliability predictions are almost valueless. Confidence-interval mathematics will work only if you have superior input data. In reality, software-development processes have too many sources of abnormal variability, especially processes that concern test time. If you were to take all these sources of variability into account to correctly compute a confidence interval, the true range would be so wide as to become pointless.

If you get bad results it's because you started with bad data. Collecting good data will be the most difficult challenge in performing reliability predictions. Good data will give good predictions which will give good correlation with actual field performance.

Good initial results will build credibility. Credibility will help get better data and so on. To get good data, software-reliability modeling must be presented as a nonthreatening tool that will help developers make development more successful.

Where is the technology going?

Everett: Any engineering discipline associated with any technology, software reliability in particular, must answer two questions: how do we measure it? and once we can measure it, how do we manage it? By managing it I mean understanding how the measures can be used to control and ultimately improve software reliability.

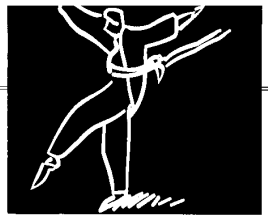
I feel the potential for the management side of software reliability can be on par with that of hardware reliability. A number of processes associated with software development can greatly affect software reliability — design reviews, inspections, requirements specification, testing, configuration control, to name a few. The challenge will be how we use reliability measures to properly manage and improve these processes, which in turn will improve the products they produce.

Tausworthe: Comprehensive modeling of the reliability process is moving toward simulating the injection and removal of faults and defects over the entire life cycle, using a sufficiently general model with parameters that adequately describe the phenomena taking place. Determining the product and process parameters of a set of actual projects and the model regression is what optimizes the performance of the comprehensive model; neural-network solutions to the regression may apply. Comparing selected prediction models using statistical data generated by a simulated reliability process may provide a controlled means to answer today's controversy over model merits.

Yu: The practical issue of software reliability is to find a mathematical function that can fit the empirical data. On the basis of the mathematical function, you can do extrapolation to predict software quality. The usefulness of theoretical work is to provide the physical meaning of the model parameters. Therefore, you can reuse the parameters of old projects when few data points are available in a new project. In addition, when the model parameters change significantly from one project to another, you can analyze the result to identify areas for process improvements.

Brettschneider: Those of us who are interested in software reliability should make every attempt to promote it — we must develop commonly available computer programs to calculate reliability predictions. Future model research should concentrate on the development of more robust models that have fewer requirements for applicability — as in a reduced need for strict formalisms — are more intuitive, and are easier to calculate and apply.

Finally, we need continued investigations into the relationships among process factors, development practices, and quality. The process of searching for the optimal model is its own chief benefit. It forces a better understanding of what process and product factors are important to reliability prediction.



because not every failure results in a defect removal and some corrections introduce defects.[4]

Defect data as predictors. Data collected in the 1970s from applying early reliability models shows that the failure rate does not remain constant — it is nonhomogeneous.[6,7] Indeed, the failure rate usually decreases as more defects are detected and corrected, contributing to the system's overall reliability.

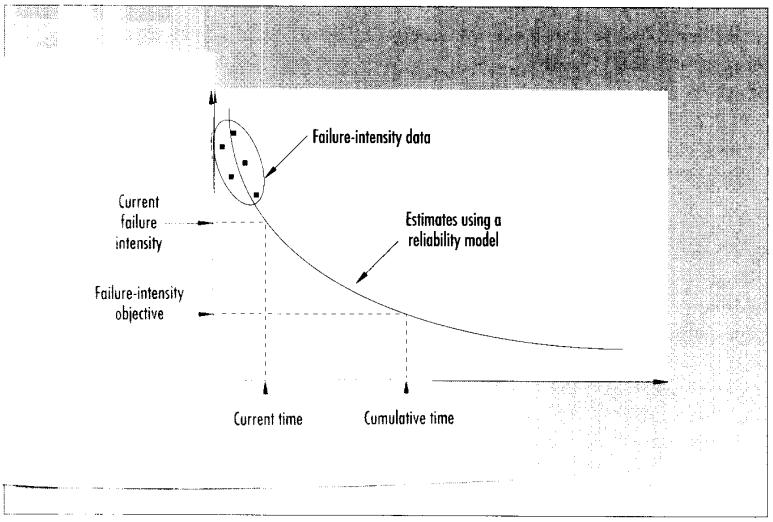


Figure 3. *Plotting defect data using a reliability model produces a failure-intensity curve. Failure intensity is the number of failures per time unit. Such a curve can help you predict the execution time needed to achieve a failure-intensity objective. Musa's calendar-time component relates execution time to calendar days, based on constraints involved in applying resources to a project.*

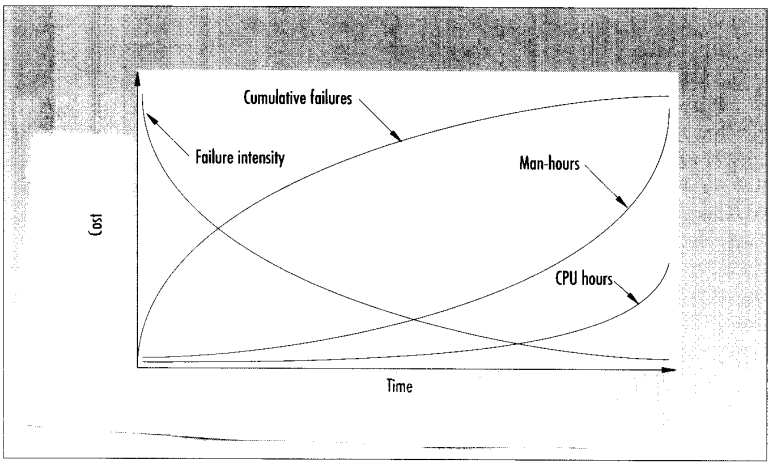


Figure 4. *As the cumulative number of failures rise monotonically over the time, the failure-intensity curve usually decreases. Typically, the failure-intensity curve for raw data (without curve fitting) will show bursts of peaks indicating increases and decreases in failure intensity. Failures tend to occur sporadically and are test-case dependent. Testing costs, however, tend to rise in a complimentary fashion: As fewer defects lurk in the code, it takes more man-hours and computer time to detect and locate them.*

Figure 3 shows a nonhomogeneous, Poisson process reliability-growth model in which failure intensity decreases exponentially with execution time, a property of John Musa's basic execution-time model.

Figure 4 shows that, in unison with decreasing failure intensity, the expected cumulative number of failures increases exponentially to an asymptote with cumulative execution time. This inverse relationship is a basis for most reliability models.

If the observed failure rate is plotted as a function of cumulative execution time, as it is in Figure 3, a reliability model can be statistically fitted to the data points. You can use the plot of the fitted failure-intensity curve to estimate failure intensity and the additional execution time required to attain the failure-intensity objective.

During this process, the developer must determine

- the estimated failure-intensity periodically during system test;
- if the estimated failure intensity is less than or equal to the objective (if it is, the software can be released), and
- if the estimated failure intensity exceeds the objective (if it does, you must identify the additional test resources needed to attain the objective).[8]

Just as failure intensity decreases exponentially, the cost of detecting a failure and locating its cause increases in a complementary exponential fashion. The cost of correcting defects, on the other hand, *generally* remains constant over time because it depends on relatively constant factors like developer expertise and tool availability.[9] Albeit, the cost of correcting defects in fielded software is higher than the cost of corrections made early in the development cycle.

Defect classification. Grouping defects into classes lets you identify their effect on the system's overall reliability. It also lets you weight them according to their criti-

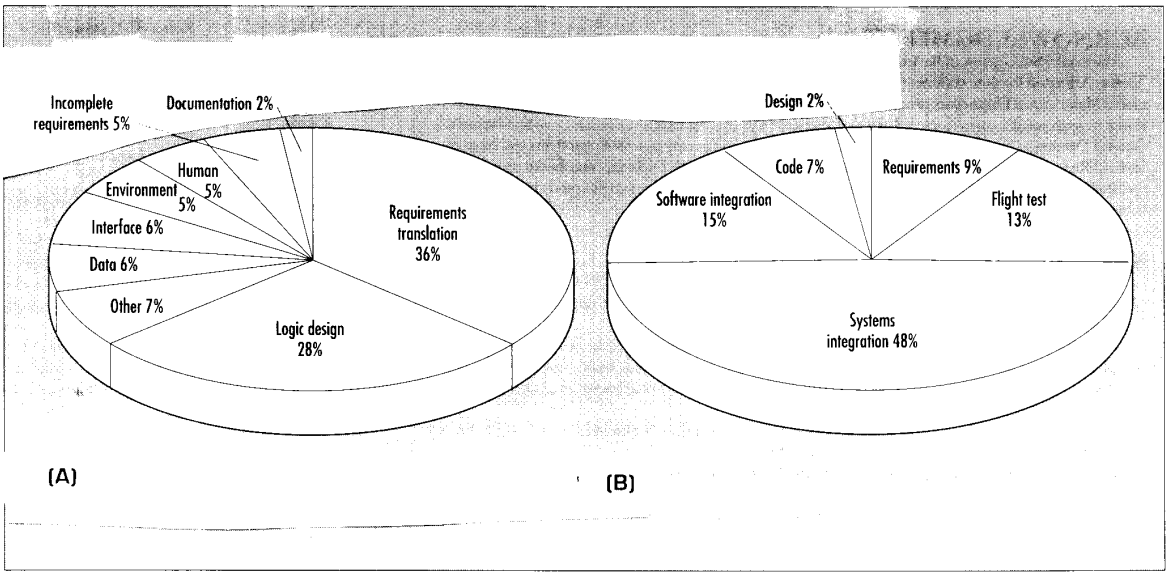


Figure 5. *As this example from a US Air Force project shows, you can classify defects by* ***(A)*** *problem type and* ***(B)*** *by the phase in the life cycle in which they are corrected. In this case, 48 percent of defects were fixed at systems integration; 13 percent at flight tests. The cost to fix defects is highest in these two phases. The software was released with six percent of problems unresolved; these problems will likely surface in the field.*

cality or severity, in terms of safety and/or customer cost. Another reason to group defects by cause is to help pinpoint where in the life-cycle they are introduced.

Classifying defects in terms of cause can help developers decide where to apply resources. Developers can improve the reliability — and hence quality — of their products by properly focusing corrective resources on the biggest problem areas.[10]

Classifying defects in terms of cause can help developers decide where to apply resources.

Figure 5a shows a sample defect-classification scheme that shows a sizeable percentage of problems occurring in requirements translation. Although the cost of detecting and removing defects is unknown, it is *thought to be* less than in later phases. As with testing, requirements checking is labor intensive, especially when only a few defects exist. In this case, targeting the requirements analysis and design phases could have a significant effect on improving the process and reducing a major problem type.

Figure 5b shows that most of the defects were not resolved until the latter development stages, when the cost of repair is the highest (as Figure 4 shows).[9]

The pie charts in Figure 5 are developer-oriented because they relate defects to development phases. In a user-oriented approach, the customer and developer together classify defects in terms of the failure or symptom as it presents itself operationally. Thus, the customer can clearly understand the level of reliability in terms of operational needs.

For example, three problem types might describe a gradient of severity from catastrophic to more benign:

- The system will not perform the tasks required.
- The system will operate in degraded mode, but with extra operational cost.
- The system will operate with minor dysfunction and small extra operational cost.

Thus, if the general reliability requirement is 1,000 hours of failure-free operation, this approach would further qualify the reliability measure to state at what level of severity failures are acceptable. This gives the customer better insight into operational expectations and promotes greater customer satisfaction. However, this developer-customer dialogue is itself a requirements-translation problem and is defect-prone.

In an environment of limited resources and tough competition, reliability measurement provides guidance to decide which known defects are most important to the customer (if your resources prohibit fixing all of them) and how to structure reliability growth testing so as to find the undiscovered defects that would most severely harm the customer. ♦

ACKNOWLEDGMENTS

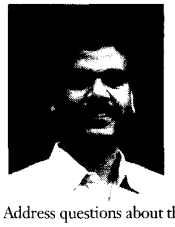
We thank Paul Grabow of Baylor University and Bill Carroll and Seung-Min Yang of the University of Texas at Arlington for their comments on early drafts. Dick Clothier from the US Air Force's Generic Integrated Maintenance Diagnostics program office and some friends at Texas Instruments tangibly illustrated reliability measurement's importance and practical application (including standardization). Dave Sundstrom of General Dynamics, Fort Worth Division, and Texas Christian University provided comments and support. Richard Reese and William Rumbley of General Dynamics, Fort Worth Division, contributed greatly to the readability. The *IEEE Software* reviewers gave us constructive comments in untangling and clarifying the contributions of many people.

REFERENCES

1. R. Brettschneider, "Is Your Software Ready for Release?" *IEEE Software*, July 1989, pp. 100-102, 108.
2. F.T. Sheldon et al., "Software Reliability Measurement Theory, Practice, and Controversy," *Proc. Symp. Applied Computing*, IEEE CS Press, Los Alamitos, Calif., 1990.
3. J.D. Musa, "Engineering Software Systems Development, Acquisition, and Use with Software Reliability Measurement," IEEE CS Press Videotapes and Video Notes, Los Alamitos, Calif., 1989.
4. Y.K. Malaiya and P.K. Srimani, *Software Reliability Models: Theoretical Developments, Evaluation, and Application*, IEEE CS Press, Los Alamitos, Calif., 1991.
5. R.C. Tausworthe, "A General Software Reliability Process Simulation Technique," Pub. 91-7, Jet Propulsion Laboratory, Pasadena, Calif., Apr. 1991, pp.1-53.
6. J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.
7. B. Littlewood, "Stochastic Reliability-Growth: A Model for Fault-Removal in Computer Programs and Hardware Designs," *IEEE Trans. Reliability*, Oct. 1981, pp. 313-320.
8. W.K. Ehrlich, S.K. Lee, and R.H. Molisani, "Applying Reliability Measurement: A Case Study," *IEEE Software*, Mar. 1990, pp. 56-64.
9. F.T. Sheldon, "Software Development and Reliability Modeling: Software Life Cycle Model," masters thesis, University of Texas, Arlington, 1988.
10. W.S. Humphrey, T.R. Snyder, and R.R. Willis, "Software Process Improvement at Hughes Aircraft," *IEEE Software*, July 1991, pp. 11-23.

Frederick T. Sheldon is a senior engineer and technical lead for Generic Integrated Maintenance Diagnostic Systems at General Dynamics, Fort Worth Division. His interests are reliability, real-time, fault-tolerant systems, computer architectures, specification, and simulation and modeling.

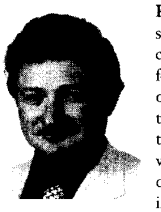
Sheldon received a BS in microbiology and a BS in computer science, both from the University of Minnesota at Minneapolis St.Paul, and an MS in computer science from the University of Texas at Arlington, where he is a PhD candidate. Sheldon received a NASA Graduate Student Fellowship from Langley Research Center, Virginia. He is a student member of the IEEE Computer Society, IEEE Reliability Society, ACM, American Institute of Aeronautics and Astronautics, the Dallas-Fort Worth Association for Software-Engineering Excellence, Tau Beta Pi, and Upsilon Pi Epsilon.



Krishna M. Kavi is a professor of computer-science engineering at the University of Texas at Arlington. His interests are computer architecture, performance and reliability analysis, formal specification and program verification, and real-time systems.

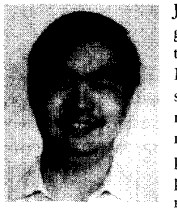
Kavi received a BS in electrical engineering from the Indian Institute of Science, Bangalore, and an MS and a PhD in computer science from Southern Methodist University. He is an editor f or IEEE Computer Society Press.

Address questions about this article to Sheldon at General Dynamics Fort Worth Division, PO Box 748, Mail Zone 2291, Fort Worth, TX 76101; Internet sheldon@cse.uta.edu.



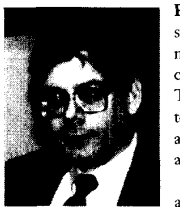
Robert C. Tausworthe is senior research engineer and chief technologist of the Information Systems Division of the Jet Propulsion Laboratory at the California Institute of Technology. His software interests are development-process modeling, simulation, and improvement. He has written the two-volume *Standardized Development of Computer Software* (Prentice-Hall, 1976, 1979), 15 papers on software methodology and analysis, and more than 100 papers on communication theory and mathematics.

Tausworthe received a BS in electrical engineering from New Mexico State University, and an MS and a PhD in electrical engineering from the California Institute of Technology. He is an IEEE Fellow, and a member of ACM and Sigma Xi.



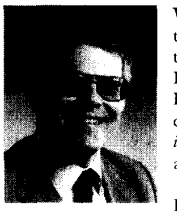
James T. Yu is a distinguished member of the technical staff at AT&T Bell Laboratories. His research interests are quality measurement and management, quality modeling and prediction, object-oriented programming, and development methodology.

Yu received a BS in electrical engineering from National Taiwan University, Taipei, and an MS and a PhD in computer science from Purdue University. He is a member of the IEEE Computer Society.



Ralph Brettschneider is software quality-assurance manager for Motorola's Microprocessor and Memory Technologies Group. His interests are managing quality assurance, metrics, and reliability modeling.

Brettschneider received a BS in biochemistry from the University of Houston and an MBA from Lake Forest Graduate School of Management, Illinois.



William W. Everett is a distinguished member of the technical staff in the Quality Process Center at AT&T Bell Laboratories. He is the coauthor of AT&T's *Reliability by Design* handbook and an editor for *IEEE Software*.

Everett received an Engineer's Degree from the Colorado School of Mines and a PhD in applied mathematics from the California Institute of Technology.