The testing of software systems is subject to strong conflicting forces. A system must function sufficiently reliably for its application, but it must also reach the market at the same time as its competitors (preferably before) and at a competitive cost. Some systems may be less market-driven than others, but balancing reliability, time of delivery, and cost is always important. One of the most effective ways to do this is to engineer the test process through quantitative planning and tracking. Unfortunately, most software testing is not engineered, and the resulting product may not be as reliable as it should be, and/or it may be too late or too expensive.

Software-reliability-engineered testing combines the use of quantitative reliability objectives and operational profiles (profiles of system use). The operational profiles guide developers in testing more realistically, which makes it possible to track the reliability actually being achieved.

AT&T was a major application field for SRET. Hence in this article, I describe SRET in the context of an actual project at AT&T, which I call Fone Follower. I selected this example because of its simplicity; it in no way implies that SRET is limited to telecommunications systems. SRET is based on the AT&T Best Current Practice of Software Reliability Engineering, approved in May 1991. Qualification as an AT&T best current practice requires use on typically eight to 10 projects with documented large benefit/cost ratios, as well as a probing review by two boards of high-level managers. Some 70 project managers also reviewed this particular practice. Standards for approval as a best current practice are high; only five of 30 proposed best current practices were approved in 1991.

**PROCESS OVERVIEW**

The standard definition for software reliability is the probability of execution without failure for some specified interval, called the mission time. This definition is compatible with that used for hardware reliability, though the failure mechanisms may differ. In fact, SRET is generally compatible with hardware reliability technology and practice. This is important because no system is purely software; all systems mix software and hardware components. A “software system” is really a “software-based system.”

**Application scope**

You can apply SRET to any software system and for most kinds of testing. The only requirement is that testing be spread broadly across system functions. For example, you can apply it to feature, load, performance, regression, certification, or acceptance testing. However, it is not appropriate for testing an isolated set of functions, such as recovery testing of a few specific capabilities. SRET should be applied over the entire software life cycle, including all releases, with particular focus on testing the phases...
Figure 1. The core application steps of software-reliability-engineered testing and the corresponding development life-cycle stages.

...from subsystem test to delivery. Testers are the prime movers in implementing SRET, but system engineers, architects, and users are also involved.

Testing types
There are two types of SRET: development testing, in which you find and remove faults, and certification testing, in which you either accept or reject the software. In many cases, the two types of testing are applied sequentially. Development testing precedes certification testing, which in turn serves as a rehearsal for customer acceptance testing.

During development testing, you estimate and track failure intensity, which is the failures per unit execution time, say, six failures per 1,000 execution hours. Execution time is the actual time used by a processor in executing a program’s instructions. Failure intensity is an alternative way of expressing software reliability. Testers use failure intensity information to determine any corrective actions that might need to be taken and to guide release. The release decisions include release from system test to beta test and release from beta test to general availability. Development testing typically comprises feature, load, and regression testing. It is generally used for software developed in your own organization.

Certification testing, on the other hand, does not involve debugging. There is no attempt to “resolve” failures you identify. Certification testing typically comprises only load testing. It is generally used for software you acquire, including off-the-shelf or packaged software, reusable software, and software developed by an organization other than your own.

Application steps
SRET has seven major steps. The first two steps consist of decision making and form the basis for the five core application steps. The decision-making steps generally take only a few hours. The core steps take considerably longer. Figure 1 shows the core steps along with the development life-cycle stages in which they typically occur. The core steps are virtually identical for the two types of testing, except for the first and last steps. I describe the core steps in more detail later in the context of the Fone Follower project.

- Determine which associated systems require separate testing. In addition to the testing of the entire system, testers may need to test major system variations. The system may have different hardware configurations, for example, or it may work with different communication protocols in different countries. You should test any major components of unknown reliability. You can also profitably test smaller components you expect to reuse extensively. If the system interacts strongly with other software systems, you may want to test a subsystem that represents these systems functioning together.
- Decide which type(s) of SRET are needed for each system to be tested. Development testing is appropriate only for systems that you code at least in part, for example.
- Define “necessary” reliability. This step consists of determining operational modes, defining failure in terms of severity classes, setting failure intensity objectives for the developed software, and engineering the reliability strategies you will employ. It does not include engineering reliability strategies for software that involves only certification testing. Although this step is performed primarily by system engineers and architects, it affects testers also.
- Develop operational profiles. An operation is a major task the system performs. Some examples of operations are a command activated by a user, the processing of a transaction sent from another system, a response to an event occurring in an external system, and a routine housekeeping task activated by your own system controller. An operational profile is simply the set of operations and their probabilities of occurrence. Activities in this step include developing two types of operational profiles (described in detail later).
- Prepare for testing. This step includes preparing test cases, test procedures, and any automated tools you decide to use.
- Execute tests. This step entails conducting testing and then identifying failures, determining when they occurred, and establishing the severity of their impact.
- Interpret failure data. The interpretation of failure data is different for development testing and certification testing. In development testing, the goal is to track progress and compare present failure intensities with their objectives. In certification testing, the goal is to determine if a software component or system should be accepted or rejected, with limits on the risks taken in making that decision. For development testing, failure data is generally interpreted at fixed intervals. For certification testing, interpretation is done after each failure.

The SRET process follows a spiral model, in a manner analogous to the software development process. Iteration occurs frequently. Figure 1 represents an “unwound coil” of this model. “Execute tests” and “Interpret failure data” occur simultaneously and are closely linked, with the rel-
ative emphasis on interpretation increasing with time.

Testers define necessary reliability and develop operational profiles in partnership with system engineers and architects. I and my AT&T colleagues originally thought that these activities should be assigned solely to system engineers. However, this did not work well in practice.

Testers depend on these activities and are hence more strongly motivated than system engineers to ensure their successful completion. We solved the problem by making testers part of the system engineering and architecture team. This approach also had unexpected side benefits. Testers had much more contact with users, which was very valuable in knowing what system behavior would be unacceptable and how unacceptable it would be, and in understanding how users would employ the product. The result was more realistic testing. System engineers and architects obtained a greater appreciation of testing and of where requirements and design needed to be made less ambiguous and more precise so that test-case and test-procedure design could proceed. System testers made valuable contributions to architecture reviews, often pointing out important capabilities that were missing.

**FONE FOLLOWER**

Fone Follower is a system that lets telephone calls “follow” users anywhere in the world (even to cellular phones). Users dial in to a voice-response system and enter the telephone numbers at which they plan to be at various times. Most of these entries are made between 7 a.m. and 9 a.m. each day.

Calls that would normally be routed to a user’s telephone are then sent to Fone Follower, which forwards them in accordance with the program entered. If there is no response and the user has pager service, the system pages. If there is still no response or if the user doesn’t have pager service, the system forwards calls to the user’s voice mail.

Fone Follower was designed to use a vendor-supplied operating system. We did not know the reliability of the operating system, so we decided to certify it. Fone Follower does not interact substantially with other systems in the telecommunications network.

**APPLYING SRET**

In the following description of how we applied SRET to the Fone Follower project, I have changed certain information to keep the explanation simple and to protect proprietary data.

**Decision-making steps**

The decision was to apply development testing to the product and certification testing to the operating system.

**Define “necessary” reliability**

As I described earlier, this step consists of four parts.

**Determine operational modes.** An operational mode is a distinct pattern of system use and/or environment that needs separate testing because it is likely to stimulate different failures. A separate operational mode may also be established to provide for accelerated testing of rarely occurring but critical operations. “Critical” in this context means that an operation adds considerable extra value when it executes successfully, or results in an extreme impact when it fails. “Value” and “impact” refer to safety with respect to human life, cost, or service.

Factors that may yield different operational modes include day of the week or time of the day (prime hours versus off hours), time of the year (year-end processing for accounting systems), traffic levels, user profile, user experience, system maturity, reduced system capability, and rare critical events. Division into operational modes is based on engineering judgment: more operational modes can increase the realism of test, but they can also increase the effort and cost of selecting test cases and performing system test.

We selected three operational modes for Fone Follower:

- **Peak hours.** Heavy incoming calls and entries (of phone numbers) traffic, no administration or audit functions permitted.
- **Prime hours.** Average incoming calls and entries traffic, administration functions permitted but audit functions limited.
- **Off hours.** Low incoming calls and entries traffic, low administration traffic, extensive audit traffic.

**DEFINE FAILURE IN TERMS OF SEVERITY CLASSES.** A failure is the departure of program behavior during execution from user requirements; it is a user-oriented concept. A fault is the defect in the program that causes the failure when executed; it is a developer-oriented concept. Thus, when you define failures you are implicitly expressing users’ negative requirements for program behavior. The definition process itself consists of outlining these negative requirements in a system-specific fashion for each severity class.

A severity class is a set of failures that affect users to the same degree. The severity is often related to the criticality of the operation that fails. Common classification criteria include impacts on human life, cost, and service. In general, classes are widely separated in impact because you can’t estimate impact with high accuracy.

We used service impact as the criterion for establishing the following severity classes in Fone Follower:

- **Class 1.** Failure prevents calls from being forwarded.
- **Class 2.** Failure prevents phone number entry.
- **Class 3.** Failure makes system administration more difficult although it remains possible through alternate means: for example, you can’t add or delete users from a graphical user interface, but you can still accomplish this with text-based commands.
- **Class 4.** Failure affects an operation that is deferrable, such as preventive maintenance.

**SET FAILURE INTENSITY OBJECTIVES.** These failure intensity objectives are for the developed software. You can choose to set them for the system or separately for each operational mode and severity class. Also, there may be different objectives for the end of system test and the end of beta test.

Setting failure intensity objectives for the developed

---

The Fone Follower system lets telephone calls “follow” users anywhere in the world.
Table 1. Part of an operational profile in tabular form.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operations/Clock hour</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect call</td>
<td>71,000</td>
<td>0.710</td>
</tr>
<tr>
<td>Phone number entry</td>
<td>10,000</td>
<td>0.100</td>
</tr>
<tr>
<td>Audit phone number database</td>
<td>900</td>
<td>0.009</td>
</tr>
<tr>
<td>Total</td>
<td>100,000</td>
<td>1.000</td>
</tr>
</tbody>
</table>

Figure 2. Graphical representation of an operational profile for a telecommunications switching system. The graphical representation is better than the tabular representation (Table 1) when there are multiple attributes.

Software has several steps:

1. Establish the system failure intensity objectives, which you derive from an analysis of specific user needs, existing system reliability and the degree to which users are satisfied with it, and the capabilities of competing systems.
2. Determine and sum the failure intensities of the acquired hardware and software components (these will often be certified at acceptance of delivery).
3. Subtract the total acquired failure intensities from the system failure intensity objectives in clock hours. This gives you the failure intensity objectives for the developed software.
4. Convert the results into failure intensity objectives for the developed software per unit of execution time.

ENGINEER RELIABILITY STRATEGIES. There are three principal reliability strategies: fault prevention, fault removal, and fault tolerance. Fault prevention uses requirements, design, and coding technologies and processes, as well as requirements and design reviews, to reduce the number of faults introduced in the first place. Fault removal uses code inspection and development testing to remove faults in the code once it is written. Fault tolerance reduces the number of failures that occur by detecting and countering deviations in program execution that may lead to failures.

Engineering these reliability strategies means finding the right balance among them to achieve the failure intensity objective in the required time and at minimum cost. In addition to achieving balance, system engineers/architects focus on operations that are most frequently executed and/or most critical.

In later product releases, system engineers/architects obtain and use information on the actual failure intensities of components in relation to their failure intensity objectives to determine which components a project should target in its reliability-improvement efforts. System and field engineers evaluate field failures in terms of their frequency and severity. They then assign priorities to the project’s responses to the failures. System engineers/architects attempt to determine how the development process could be cost-effectively improved to avoid the most important field failures.

Develop operational profiles

You will need to develop two kinds of operational profiles, overall (across all operational modes) and operational mode. The overall profile is used to select the test cases to prepare. The operational profile for each operational mode is used to select operations for execution when that mode is tested.

Fone Follower expected an average of 100,000 operations per clock hour over all operational modes. Table 1 shows a segment of its operational profile and how we obtained it from the occurrence rates of individual operations. This is a tabular representation. You can also represent the operational profile graphically as a network of nodes and branches. The tabular representation is generally better for systems whose operations have few (often one) attributes. The graphical representation is generally better for systems whose operations have multiple attributes, such as telecommunications switching systems. The operations of Fone Follower had only one attribute, so we chose the tabular representation.

For the sake of illustration, Figure 2 shows part of a graphical representation of a telecommunications switching system. The nodes represent attributes of an operation; the branches represent values. An operation is represented by a path through the network. The occurrence probabilities of attribute values can be conditional on the previous path to that point.

Regardless of which representation you choose, the procedure to develop an operational profile is similar. Here is the procedure for tabular representation (with the variations for graphical representation in parentheses):

- Identify the initiators of operations. System users are most commonly the initiators of operations, but initiators can also be external systems and the system's controller. To identify the users, you first determine the expected customer types on the basis of information such as the system business case and marketing data for related systems. You can then analyze the cus-
Table 2. Levels of the four direct input variables for the connect call operation of Fone Follower.

<table>
<thead>
<tr>
<th>Call type</th>
<th>Destination</th>
<th>Billing type</th>
<th>Dialing type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fax</td>
<td>Local calling area</td>
<td>Flat rate</td>
<td>Standard</td>
</tr>
<tr>
<td>Voice</td>
<td>Within area code</td>
<td>Per call</td>
<td>Abbreviated</td>
</tr>
<tr>
<td>Data-modem</td>
<td>Outside area code</td>
<td>Per-call discount</td>
<td></td>
</tr>
<tr>
<td>Data-ISON</td>
<td>International Wireless</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Prepare for testing

The activities in this step consist of specifying the test cases and test procedures and preparing any automated tools you decide to use. For these activities, it is worth noting some important definitions.

A run is a specific instance of an operation and is characterized by that operation and a complete set of values of its input variables. Input variables are variables that exist external to the run and influence it. In Fone Follower, for example, a run for the connect call operation is specified by seven input variables: call type (for example, fax), dialing type (standard or abbreviated), destination, billing type, operational mode, database state, and resource state.

The first four input variables are direct because they control processing in a known, designed way. The last three are indirect because although they influence processing, it is not in any predictable way. For example, if an operational mode has heavy traffic, it may change processing because of resource conflicts. A different database state, due to, say, increasing data corruption with time, may also change processing as a result of its different values.

A run differs from a test case. In a test case, you provide the values of only direct input variables. A test case becomes a run when you also specify the values of indirect input variables. This has implications for testing productivity because, by changing the indirect input variables, testers can use a moderate number (perhaps hundreds) of test cases to generate a very large number of different runs.

**Specify test cases.** The first activity is to estimate the number of test cases you can cost-effectively prepare, on the basis of the effort needed for each. You can then select test cases in two steps.

1. Select the operations in accordance with their occurrence probabilities, using the overall operational profile, modified appropriately for any critical operations. For Fone Follower, occurrence probability of the connect call operation was 0.71 (Table 1), so 71 percent of the test cases were from the connect call operation.
2. Complete the selection of the test case by choosing levels for all direct input variables. A level is a value or a range of values of an input variable for which failure behavior is expected to be the same because of processing similarities. For example, in Table 2, a value of the input variable “Destination” would be a specific phone number. “Local calling area” would be a level of that input variable because the system would probably behave in the same way for all local phone numbers. You need select only one value within a level for a test case. In some cases, a level has only one value, such as “fax” in “call type.”

Step 2 has several activities. The first is to list the levels for each direct input variable, as we have done in Table 2 for the connect call operation. The next activity is to randomly choose a level for each direct input variable. The choice should be made with equal probability from the set of levels. For the connect call operation, a possible random selection of levels is “Fax, outside area code, per call, abbreviated.” The operation and the set of direct input variables selected specify a test case. The last activity is to write a test script to implement the test case selected.

**Define test procedure.** A test procedure is the specification of the set of runs and environment associated with an operational mode. You generally specify the set of runs statistically by providing values of operation occurrence rates. When executed, the test procedure will select test cases at random times from the prepared set, following these occurrence rates.

In feature testing, test runs are executed essentially independently of each other, with the database reinitialized frequently to reduce interactions. In load testing, large numbers of test runs are executed in the context of an operational mode, driven by a test procedure. Load
Execute tests

During this step, you begin with feature testing and follow that with load testing. Regression testing is typically done after each build that involves significant change. In load testing, you execute each operational mode separately. For Fone Follower, this means we tested the peak hours, prime hours, and off hours operational modes. You should allocate execution time among the operational modes for testing in the same proportions that they are expected to have in the field, adjusting for critical operational modes.

Test execution involves identifying failures, determining when they occurred, and establishing the severity of their impact. To identify failures, first look for deviations of program behavior. Determine which ones affect the user; only those that do are failures. Generic tools can detect many types of deviations, including interprocess communication failures, illegal memory references, and deviant return code values. You can also manually insert assertions in the code to detect programmer-defined deviations. You will probably require at least some manual inspection of test results to identify failures not amenable to automatic detection and to sort out deviations that are true failures—unless you can demonstrate that the ratio of failures to deviations is essentially constant. In that case, you can use deviation data in place of failure data, adjusting by the known ratio. You must also consider many special circumstances such as cascaded failures, repeated failures, and failures that will deliberately not be resolved.  

You should determine the time of failure occurrence or number of failures per time period in execution time. If execution time is not readily available, you can use one of many possible approximations. You will need a special approach to handle failure data if testing is done on multiple machines.  

Interpret failure data

The interpretation of failure data depends on whether you are doing development or certification testing.

Development testing. The first activity is to consider trends (in feature, load, and regression testing) by estimating and plotting the failure intensity over all severity classes and across all operational modes against calendar (ordinary) time. Figure 3 shows a failure intensity trend very similar to what we obtained for Fone Follower. The center plot (black) represents the most likely estimate; the other two plots (gray) represent the upper and lower 95-percent confidence bounds. Comparison with the overall failure intensity objective helps you identify “at risk” schedules or reliabilities and lets you take appropriate and timely corrective actions. Long, large upswings in failure intensity commonly indicate that either the system has evolved or the test operational profile has changed. System evolution can be an indication of poor change control. A change in the test operational profile may indicate poorly planned test execution. In either case, corrective actions are necessary if you are to have a dependable test.

The next activity is to guide release decisions. If the failure intensity objectives were set by severity class by operational mode, you should estimate the failure inten-
Software reliability engineering: Rapidly growing discipline

Software reliability engineering is maturing to the extent that standards bodies and publishers are undertaking activities to promote it. An AIAA standard for software reliability engineering was approved in 1993, for example, and IEEE standards are under development. McGraw-Hill and the IEEE Computer Society Press recently recognized the rapid maturing of the field, publishing a handbook on the topic. The IEEE Computer Society's Technical Committee on Software Reliability Engineering has grown in the six years since its founding from around 40 people to more than 1,000, an annual growth rate of about 70 percent. It publishes a newsletter, sponsors the annual International Symposium on Software Reliability Engineering, and engages in numerous other activities, including standards.

Growth of the software reliability engineering research community is about 35 percent per year, as judged by the number of papers submitted to ISSRE. There is also an active software reliability engineering bulletin board on the Internet.

Success stories

The growth of software reliability engineering is not unfounded. Those incorporating it into their development practices are enjoying significant improvement. In AT&T's Operations Technology Center of the Network Computing Services Division, software reliability engineering is part of the standard software development process, which is currently undergoing ISO certification. This organization, which has the highest percentage use of software reliability engineering in AT&T, is the primary software development organization for the AT&T business unit that won the Malcolm Baldrige National Quality Award in 1994. In addition, four of the first five software winners of the AT&T Bell Laboratories President's Quality Award used software reliability engineering.

AT&T's International Definity project illustrates the benefits that result from applying software reliability engineering along with related technologies. In comparison with a previous release that did not use these technologies, reliability increased by a factor of 10. Customer satisfaction with the product increased correspondingly—sales increased by a factor of 10. Moreover, both the system test interval and system test costs decreased by a factor of two, the total project development time decreased by 30 percent, and maintenance costs decreased by a factor of 10.

These successes are not limited to AT&T, Alcatel, Bellcore, Bell-Northern Research (Canada), CNET (France), ENEA (Italy), Ericsson Telecom, Hewlett-Packard, Hitachi, IBM, Jet Propulsion Laboratory, Loral, Microsoft, Mitre, Tandem Computers—just to name a few—have also used software reliability engineering profitably.

As benefits like this become more widely publicized, software reliability engineering can be expected to grow even more. Many areas that are as yet untouched might benefit greatly from this discipline. In object-oriented technology, for example, there seems to be potential in applying software reliability engineering to certify object libraries. Although object-oriented concepts have made better modularity possible, the promise and benefits of reuse are not being fully realized because developers (and probably rightly so) strongly resist using objects whose reliability they cannot vouch for.

Becoming involved

Those interested in learning more about software reliability engineering have many opportunities for participation. To join the IEEE Computer Society Technical Committee on Software Reliability Engineering (membership is free), contact http://www.cs.org/tcseform.html. To subscribe to the software reliability engineering bulletin board on the Internet, contact vishwa@hac2arpa.hac.com; to post items on the bulletin board, contact sw-rel@igate1.hac.com.

Reference


You can then plot each failure on the chart and label it with its severity class. Depending on the region in which it falls, you may accept or reject the software being tested or continue testing. In Figure 4, the testing resulted in three failures, which occurred consecutively in time. Each of the first two indicates that testers should continue testing. The third falls in the accept region, indicating that sufficient data has finally been collected to demonstrate that the software can be accepted at the risk levels for which the chart was constructed.

You can build reliability demonstration charts for different levels of consumer risk (the risk of accepting a bad program) and supplier risk (the risk of rejecting a good program).

Practitioners have generally found SRET unique in offering a standard proven means to engineer and manage systems that way and compare them with the failure intensity objectives you have chosen. If there is only an overall failure intensity objective, you need estimate only an overall failure intensity and compare it with the objective. Release decisions include component test to system test, system test to beta test, or beta test to general availability.

You can estimate failure intensity from failure times or the number of failures per time period, using reliability estimation programs based on software reliability models and statistical inference, such as CASRE (Computer-Aided Software Reliability Estimation).

Certification testing. Certification testing uses a reliability demonstration chart, such as that in Figure 4. Pone Follower applied a similar chart to certifying its operating system. To normalize failure times, you multiply them by the appropriate failure intensity objective.

November 1996
age testing in a way that lets them increase their confidence in the reliability of the software-based system they delivered. They have also appreciated the decrease in time to market and increase in testing efficiency. Given these reactions, I expect the use of SRET to grow in the same way that the slightly more inclusive discipline of software reliability engineering has been growing (see the “Software reliability engineering” sidebar).

This growth gives rise to the important issue of how to effectively deploy SRET. I and my colleagues have tried several approaches and have found marked differences in their effectiveness.

- **Consultant does all.** Hiring a consultant to do everything is the most expensive and probably the worst approach; true commitment of development projects is often lacking and company personnel do not gain skills in SRET.

- **Self-teaching.** This is only slightly better than having a consultant do everything. The main drawback is that project personnel have no one to answer their questions or give them feedback on what they are doing. Consequently, there is a high risk that they are not applying SRET correctly.

- **Course without a workshop.** Participants acquire knowledge, but they don’t get the opportunity to try what they have learned or to receive feedback.

- **Course with a workshop.** This approach is probably the most cost-effective one. It teaches participants the necessary skills, allows them to try out what they’ve learned and get feedback, and to ask questions. The course with the workshop is most effective when done on site, with work groups organized by projects. Each project team brings its requirements, specification and architecture documents. Each should leave with the initial outline of a plan for applying SRET to their project.

- **Custom or group jump starts.** Jump starts also teach participants the necessary skills, allow them to try out what they’ve learned and get feedback, and ask questions—but they cost considerably more than a course with a workshop. In a jump start, a consultant participates in projects over a substantial period of time to answer questions and review the work of the projects. The custom jump start, which is dedicated to one project, is more expensive than the group jump start, in which consulting resources are shared.

Jump starts are very nice for projects that can afford them, but they aren’t essential. You can choose the course with a workshop and supplement it with access to a consultant for questions without the investment in time that project participation involves. The principal high-cost element is the time the consultant must spend to learn and follow the project. Organizations can avoid this cost by selecting a project representative to work with the consultant. The project representative attends the course with a workshop along with the project team. The representative then translates project issues into questions that relate to SRET practices and poses them to the consultant.

### Acknowledgments
I am most grateful for the careful review and helpful experience-based comments of Wills Ehrlich, Susie Hill, Steve Meyer, and Ray Sandfoss of AT&T.

### References

*John D. Musa* teaches courses and consults in software reliability engineering and testing. He has been involved in software reliability engineering since 1973 and is generally recognized as one of the creators of that field. Recently, he was Technical Manager of Software Reliability Engineering at AT&T Bell Laboratories, Murray Hill. He organized and led the transfer of software reliability engineering into practice within AT&T, spearheading the effort that defined it as a "best current practice." Musa has also been actively involved in research to advance the theory and practice of software reliability engineering. He has published more than 80 articles and papers, given more than 150 major presentations, and made several videos. He is principal author of *Software Reliability: Measurement, Prediction, Application* (McGraw-Hill, 1987).

Musa received an MS in electrical engineering from Dartmouth College. He is listed in Who's Who in America and American Men and Women of Science. He is a fellow of the IEEE and the IEEE Computer and Reliability Societies and a member of the ACM and ACM Sigsoft.

Contact Musa at 39 Hamilton Rd., Morristown, NJ 07960; j.musa@ieee.org. For SRET course information, contact http://members.aol.com/JohnDMusa/Musa.html.