The Future Engineering of Software: A Management Perspective

Victor R. Basili, University of Maryland
John D. Musa, AT&T Bell Laboratories

There are many perspectives from which to view the future of software. This article focuses on the engineering process that underlies software development. This process is critical in determining what products are feasible. We support a quantitative approach and believe that software engineering must move in this direction to become a true engineering discipline and to satisfy the future demands for software development. Further, we want to spotlight some areas of software engineering that we believe have received less attention than they merit. We begin with a brief summary of how information technology has affected both institutions and individuals in the past few decades.

In the 1990s, market forces will drive software development into quantitative methods for defining process and product quality.

The past. In the 1960s, information technology penetrated institutions. This decade could be called the functional era, when we learned how to exploit information technology to meet institutional needs. Institutional functions began to interlink with software.

In the 1970s, the need to develop software in a timely, planned, and controlled fashion became apparent. This decade introduced phased life-cycle models and schedule tracking. It could be called the schedule era.

The 1980s might be named the cost era. Hardware costs continued to decrease, as they had from the early days of computing, and the personal computer created a mass market that drove software prices down as well. In this environment, information technology permeated every cranny of our institutions, making them absolutely dependent on it. At the same time, it became available to individuals. Once low-cost applications became practical and widely implemented, the importance of productivity in software development increased substantially. Various cost models came into use and resource tracking became commonplace.

The problem with these approaches to software development is their focus on single, isolated attributes. We did not understand the relations among functionality, schedule, and cost well enough to control trade-offs. We did not effectively define other attributes such as reliability, necessary for engineering a software

0018-9162/91/0900-0090$01.00 © 1991 IEEE

product that satisfies a user's needs. We did not learn enough about how to engineer and improve products based on experience.

The future. We believe the 1990s will be the quality era, in which software quality is quantified and brought to the center of the development process. This new focus on quality will be driven by the dependence of institutions on information processing. We can also expect software vendors to try to create new demand in the consumer mass market. Thus, home applications might expand rapidly in the nineties, although the time required for cultural acceptance of some applications could delay this past the turn of the century.

The consumer mass market, with its potential for large sales but rather unsophisticated users, increases the demands on quality. These demands, when satisfied, intensify competition in the institutional market because institutions can improve quality for their customers if better quality is available in the information systems they depend on. In the future, this overall intense competition will be international.

In this article we discuss software quality, software engineering that uses models and metrics to achieve quality, the processes needed to achieve software quality, and how to put these processes and technology into practice.

Software quality

Quality is not a single idea, but a multidimensional concept. The dimensions of quality include the entity of interest, the viewpoint on that entity, and the quality attributes of that entity. Entities include the final deliverable, intermediate products such as the requirements document, and process components such as the design phase. Example viewpoints are the final customer's, the developing organization's, and the project manager's. The quality attributes that are relevant in a given situation depend on both the entity and the viewpoint. For example, readability is an important quality attribute of a requirements document from the designer's viewpoint. Elapsed time is an important quality attribute of the design phase from the project manager's viewpoint. It is important to quantify these dimensions wherever possible.

Software quality attributes are not independent; they influence each other.

Product quality. The ultimate quality goal is user satisfaction. Therefore, we will consider quantitative specification of final product attributes that satisfy explicit and implicit user needs. The attributes most often named as significant are functionality, reliability, cost, and product availability date. Reliability often ranks first.

It is possible to reduce the list of attributes to three by taking a broad view of reliability as the probability (over an appropriate time period) that the product will operate without user dissatisfactions (denoted "failures"). In this view, if a function is missing when the user needs it, the event marks a failure. Thus, the attribute of functionality folds into the attribute of reliability.

The degree of quality is the closeness with which the foregoing attributes meet user needs. Competition makes it necessary to improve the match. It will increasingly force joint supplier-user setting of objectives and measurement to compare them with the results.

Software quality attributes are not independent; they influence each other. If we think of reliability in terms of failure intensity or failures per unit time, we can define a quality figure of merit as the reciprocal of the product of failure intensity, cost, and development duration. (The real relationship among the factors is probably somewhat more complex than simply taking their product, but this formulation will serve our purposes here.)

The quality figure of merit always characterizes the state of the art. Consequently, a lower failure intensity (increase in reliability) will generally require an increase in cost or development duration or both. As technology advances, the quality figure of merit increases and the lower failure intensity is achieved at lower cost or less development time or both.

Process quality. Meeting quality objectives in the delivered product requires a suitable quality-oriented development process. You can view this process as a series of stages, each with feedback paths. In each stage, an intermediate supplier develops an intermediate product for an intermediate user — the next stage. Each stage also receives an intermediate product from the preceding stage. Each intermediate product will have certain intermediate quality attributes that affect the quality attributes of the delivered product, but are not necessarily identical to them. For example, in the design stage, designers are the users for the requirements specification. They develop the system architecture and unit specifications, defining them in a design document, which is their intermediate product. Important quality attributes of the design document are readability and completeness in meeting system requirements.

In addition to viewing the development process as a series of stages with intermediate products, we need to look at it as a semistructured cognitive activity of a social group. Human cognitive processes and social dynamics in software development affect product quality. For example, some evidence suggests that informal communication networks have much more impact than documents in the software development process.

We need models of the development process, measures of its characteristics, and practical mechanisms for obtaining those measures. We need to relate the measures to the quality attributes of the deliverable product. Then, we can control the development process and adjust it to meet the attribute objectives. For example, what are the appropriate methods for developing a product that must have high reliability and what leeway in cost or delivery is permissible to achieve it?

Finally, we need models of how users will employ the system and of the relative criticality of the various operations in this context.

Engineering with models and metrics

We have had quantitative approaches to the design and implementation of pure hardware systems for some time. Scheduling, cost estimation, and reliability technologies for hardware were

fairly well developed by the 1960s, but similar technologies for software have lagged by 20 to 30 years. We believe this is due to lesser understanding of software development and the essential differences between hardware and software engineering (for example, the differences between production and development).

In spite of the complexity of the task, we must model, measure, and manage software development processes and products if we are to optimize the balance among quality attributes and satisfy user needs. Understanding where the time and effort are going and what processes provide the attributes needed for a more reliable product will help us refine models of quality attributes and the interrelationship between process and product.

To do this we must isolate and categorize the components of the software engineering discipline, define notations for representing them, and specify interrelationships among them as they are manipulated. The discipline's components consist of various processes and process components (for example, life-cycle models and phases, methods, techniques, tools), products (for example, code components, requirements, designs, specifications, test plans), and other forms of experience (for example, resource models, defect models, quality models, economic models).

We need to build descriptive models of the discipline components to improve our understanding of

(1) the nature and characteristics of the processes and products,
(2) the variations among them,
(3) the weaknesses and strengths of both, and
(4) mechanisms to predict and control them.

We have models for some components. For example, there are several mathematical models of programs and modules, such as predicate calculus, functions, and state machines.

Cost and schedule models have moved from research and development into application. There are parameterized cost models for using historical data to predict the project costs. For example, many organizations are using or studying cost models like Cost Constructive Model (Cocomo),[1] Software Life-Cycle Management (Slim), Software Productivity, Quality, and Reliability Model (SPQR), and Estimacs.[2]

Software reliability engineering models are coming into practice.[3] The exponential and logarithmic nonhomogeneous Poisson models are the most widely used models in the industry today. Japan has made some use of S-curve models.

We have models and modeling notations for various life-cycle processes. These key modeling technologies form the basis for a quantitative approach to the engineering of software-based systems — enough to start the advance of software engineering from craft to science.

However, many more areas require models. For example, little work has been done in organizing and systematizing the practical knowledge accumulating in various application domains.

We need to screen the models that do exist. They often require more formal definition, further analysis, and integration to deepen our understanding of their components and interactions. We need to eliminate models that are not appropriate or useful.

Based upon analysis of these descriptive models, we must build prescriptive models that improve the products and the processes for creating them. Prescriptive models must relate to quality attributes. We must provide feedback for project control and learn to package successful experience.

Because the overall solutions are both technical and managerial, model-building requires the support of many disciplines. The next several sections focus on areas of technology that we believe will play an important role in deepening our understanding and attainment of software quality in the next decade.

Formal methods. To improve our understanding of the software product itself and to enable the abstraction of its functionality, computer scientists have developed product models based on mathematical formalisms. These formalisms include predicate calculus, functions, and state machines (based on the work of R. Floyd, E. Dijkstra, C. Hoare, H. Mills, and others). These models have had theoretical value for many years, but they have not been used effectively in practice. This is largely due to the inability to scale them up to reasonable-size systems.

We are now beginning to see some practical application of formal methods in software development (for example, the Vienna Development Method, Z, and Cleanroom). They also may add to the associated discipline of correctness-oriented development. For an introduction to formal methods, see Wing.[3]

Design methods. The 1980s brought a major breakthrough in software design with the introduction of object-oriented design methods, technologies, and languages.[4] This approach will continue to have a major effect on software design in the 1990s. For example, we expect object-oriented technologies to play a major part in the definition of integrated support environments. The notion of managing and designing systems by objects will be better defined and will change the way we think about systems. Object-oriented approaches, like functional decomposition approaches, will become part of the software engineer's set of intellectual tools. They have already begun to affect the creation of reusable software, and this effect is expected to increase as we learn more about software engineering and reuse.

Programming languages. Languages that support object-oriented design and programming, in whole or in part, will continue to evolve (for example, Ada, Objective C, C++, Smalltalk). Notations will also evolve for formalizing higher level abstractions, such as requirements and specifications. The higher the level of these languages, the more likely they will become application oriented and specialized. For example, we will continue to see fourth-generation languages introduced for specific applications; we will also see more effective translation of these higher order languages into executable forms. These notations will become basic tools in the engineering process for software.

Measurement approaches. Measurement is associated with modeling. We must base measures on models to determine if they are performing as planned. In the past, measurement has been metric oriented, rather than model oriented. In other words, it has involved collecting data without an explicit goal, model, and context. For example, in analyzing a test process, project managers may collect data such as program size or number of defects. But they may be unable compare the data to other

projects unless the models used to specify the size and defect measures are documented with sufficient contextual information to interpret the data.

We have begun to see more organized approaches to measurement — approaches based on models and driven by goals.[5] These approaches integrate goals with models of the software processes, products, and quality perspectives of interest. They tailor these goals and models to the specific needs of the project and the organization. For example, if the goal is to evaluate how well a system test method detects defects, then models of the test process and defects must be available. Information that supports interpretation must be collected and integrated. For example, how effectively was the test method applied? How well did the testers understand the requirements? How many failures occurred after system test compared with similar projects?

Mechanisms for defining measurable goals have come into use. These include the goal/question/metric paradigm, the quality function deployment approach,[6] and the software quality metrics approach.[7] We expect the use of these frameworks to increase in the future.

Usage and reduced-operation software. Software usage will guide software development. An operational profile, the set of expected user operations and their probabilities of occurrence, will be defined at the same time as the system requirements. Operations are akin to functions except that they also incorporate the concept of the environment. Operations are classified by criticality where appropriate. The operational profile, adjusted for criticality, will guide the setting of priorities and allocation of effort for the entire development process.

There is an excellent chance that we will see the emergence of reduced-operation software. ROS is the software analog of reduced instruction-set computing. It is based on the observation that most software has a few operations that are used most of the time and many operations that are used rarely. The rarely used operations eat up a large proportion of development, documentation, and maintenance costs. They also complicate the system, making user training much more difficult. The ROS approach avoids implementing as many of the rarely used operations as possible. In many cases, they can be replaced by sequences of more basic, frequently used operations. System and software designers might set up these sequences and document them for users or leave them for users to determine, since many users may never require them.

We are likely to see more focus on the internal problem-solving activity of individuals and on ways to enhance its quality.

Reuse. In the past, reuse was limited mostly to the code level and based on individual experience. Interest and technology development have recently surged in this area. We can and must reuse all kinds of software experience, but reusing an object requires the concurrent reuse of the objects associated with it. For example, we have seen the development of faceted schemes, templates, and search strategies associated with reusable software components. Objects may have to be tailored for a particular project's needs; hence, reusable objects must be evaluated for reuse potential and processes must be established for enabling reuse.

Reuse will grow in the next decade based on better understanding of its implications and on development of supporting technology. Object-oriented design should make reuse easier.

Cognitive psychology. Cognitive psychology is the study of problem solving. We can use its disciplines to study different intellectual activities in the software development process. To date, very little research based on cognitive psychology has been performed in software engineering, but there is substantial evidence of its promise. Software engineering, after all, is primarily a problem-solving activity. Unfortunately, few researchers are trained and experienced in both fields. The two fields also have significant cultural differences, which can make cross-fertilization difficult. For example, many software researchers pride themselves on the controlled discipline and logic they believe is central to their approach to problems. Cognitive psychologists sometimes focus on the deficits and weaknesses they find inherent in all human intellectual processes.

Application of cognitive psychology in software engineering has generally focused on the human-computer interface. This focus is implicit in computer-aided software engineering tools. CASE has championed such human-computer interface design principles as protecting users from mistakes, helping them navigate easily through the commands and data, providing for direct manipulation of objects (for example, screen editors), and using metaphors (for example, the "sheets of paper" metaphor of windows).

This tools-oriented work will undoubtedly continue, but we are likely to see more focus on the internal problem-solving activity of the individual and the methodologies and environmental factors that can enhance the quality and efficiency of this activity. For example, people are known to have limited short-term memory. Are there software development methods that deal with this limitation in a way that increases programmer productivity? Does this limitation tend to produce certain types of faults? If so, can we use that information to improve reliability and debug more efficiently?

Early work of Curtis, Krasner, and Iscoe[8] indicates that application domain knowledge is a principal factor in the wide performance differences among software developers. This belies the frequently held concept that software development is a domain-independent activity that can be abstracted and taught totally by itself. The findings argue for a certain degree of specialization among programmers. Attention must be given to organizing, publishing, and advancing knowledge in specific domains and to providing corresponding education, either formal or on-the-job.

A study of expert debuggers[9] shows that the stereotype of these people as isolated software "freaks" is not true. The best debuggers have excellent communication, negotiation, team building, and other social skills. They generally have a clear vision of the system's purpose and architecture. They typically cultivate an extensive network of experts they can call on. The career importance of these social skills indicates that education in these areas should

start in the university and continue in the workplace.

Software sociology. Most software projects are group activities, involving all the complexities of group dynamics, communication networks, and organizational politics. The study of group behavior in software development is in its infancy, but like the study of individuals, it promises to improve our understanding of the development process, particularly at the front end. Many observers believe that improving this phase of development could have the most impact on software quality and productivity.

Software developers commonly face inefficiencies and quality degradation that result from highly volatile requirements. Some change is unavoidable because user requirements evolve with time. However, poor communication accounts for much of this problem. Research on this problem[8] has shown that successful software development is a joint process in which the developer learns the application domain and user operations, and the user learns the design realities and available choices.

Negotiation and conflict resolution are inescapable parts of the process. Managing the learning and negotiation processes intelligently is critical to success. So is making decisions in a timely fashion. In fact, there is some indication that the percentage of unresolved design issues at a given point in the project life cycle may be a good indicator of progress and predictor of future trouble. Measures can play an important role in making the negotiation process concrete and the negotiated agreements specific.

Inadequate documentation has been blamed for many project problems that appear to stem from poor communication. However, documentation may not be the real culprit.[8] Many developers do not consider it possible to maintain documentation that is sufficiently current to meet their needs. They get their information through informal networks. This suggests that we devote more effort to encouraging, cultivating, maintaining, and supporting such networks.

Improving software quality

Engineering processes require models of the various entities within a discipline. The models must approximate reality and include a controlled feedback loop to monitor the differences between the models and reality. In software engineering, we have often not had enough models to complete the process. Where we do have models (for example, for costs and schedules), we do not sufficiently understand the relationship between them and the other discipline entities.

Models are necessary for focusing attention on the multiplicity of issues necessary for engineering a product. But so is a process that supports feedback, learning, and the refinement of the models for the environment.

Manufacturing has learned to control production using models and measurements of the process and product. Feedback processes, such as the Plan-Do-Check-Act cycle,[10] have provided quality-oriented processes for manufacturing. The Deming paradigm uses models and measures to control and engineer the characteristics of processes and products. The Plan phase sets up measures of quality attributes as targets and establishes methods for achieving them. The Do phase produces the product in compliance with development standards and quality guidelines. The Check phase compares the product with the quality targets. During the Act phase, problem reports become the basis for corrective action. Achieving the quality target is the gate to the next phase.

Although software development is fundamentally different from manufacturing, at some level the same principles apply. We need a closed-loop process with feedback to the project and the organization. The process must consider the nature of software development. The quality improvement paradigm[5] is an example approach. It can help in applying, evolving, tailoring, and refining various models and ranges of measurements in software development.

In QIP, *planning* requires models of the various software products and product quality attributes, processes and process quality attributes, and environmental factors. The models must be quantifiable and the measures for them must be set. Developers must understand particular project needs with respect to such factors as functionality, schedule, cost, and reliability. Project and corporate goals are set relative to measurements associated with the models. Unlike manufacturing, there is no single model of the process. Developers must choose the process to meet the mix of quality attributes required by a particular product's user.

Doing and *checking* require following the selected process and taking measurements to track conformance with the models. Because many models are primitive, we also need to track whether the model's predictions are valid and, if they are not, modify the models to come closer to reality.

Acting requires a closed-loop project cycle with feedback for modifying models as well as the processes. It involves analyzing and packaging the experience gained on a project so that it is available to other projects. Analysis includes a postmortem review of the feedback data to evaluate the existing models, determine problems, record findings, and recommend future model improvements. Packaging involves implementing model improvements and storing the knowledge gained in an experience database available for future projects. This represents a closed-loop organization cycle that transfers learning from project to project.

Emphasis on the engineering process will help achieve quality goals for software development. It also supports the transfer of technology within and from outside an organization.

Making software engineering technology more transferable

Transferring technology within an organization requires an evolutionary, experimental approach, similar to QIP. Such an approach bases improvements in software products and processes on the continual accumulation of evaluated experience (learning) in a form that can be effectively understood and modified (such as with experience models). Experience models must be integrated into an experience base that can be accessed and modified to meet the needs of new projects (reuse). The evolving process and product models can help in technology transfer. They represent what we know and can apply in the software development process.

This paradigm implies the separation of project development, which we can assign to a project organization, from the systematic learning and packaging

of reusable experiences, which we can assign to a so-called *experience factory*.[5] The project organization's role is to deliver the systems required by the user, taking advantage of whatever experience is available. The experience factory's role is to monitor and analyze project developments; to package experience for reuse in the form of knowledge, processes, tools, and products; and to supply these to the project organization upon request.

In this sense, the experience factory is a logical organization, a physical organization, or both. It supports project developments by acting as a repository for experience, analyzing and synthesizing the experience, and supplying it to various projects on demand. The experience factory evaluates experience and builds models and measures of software processes, products, and other forms of knowledge. It uses people, documents, and automated support to do so.

Transferring software technology into an organization

There has been little success to date in transferring new software engineering methodologies and tools into active practice, despite the potential benefits of improving the development process in an industry of software's size and importance. This failure may be partly because software engineering is a process rather than a product. It is an abstract intellectual activity with limited visibility, which makes it much more difficult to transfer.

Also, research and practice in software engineering have been divided, both organizationally and by cultural values, impeding good communication.

Most practicing software engineers are not aware of all the possibilities for improvement that exist. Most researchers are not aware of the full range of problems that must be solved before new technologies can be applied in practice. Tools and methodologies are often difficult to learn or to use or both. Although most people realize that improvement means change in practice, few have been willing to deal with the cultural, motivational, and other factors that impede change.

The situation is not likely to change until researchers and practitioners deal explicitly with these factors, rather than leaving them to chance.[11] They must support change proactively.

Research and practice in software engineering have been divided, both organizationally and by cultural values, impeding good communication.

Practitioners need to address the requirements and possibilities for improvement. Improvement requires widespread education in new methods, closely integrated with education in tools. We need methods and tools that are easy to learn and use. The new technology must evolve and adapt as we gain experience with its use and continually evaluate its successes and failures. A concept like the experience factory can help with this.

A planned approach is necessary. Strategic planning identifies the goals for change and provides a basis for continuing evaluation of activities undertaken to achieve the goals. In general, this approach follows the classic phases of technology transfer: raising awareness, cultivating interest, and persuading someone to try the technology, followed by trial use and full adoption. Raising awareness typically involves the use of publications, talks before organizations, videos, and demonstrations at exhibits and meetings.

Market research is important in finding connections between project requirements and the opportunities offered by new technology. Then, technical research can find solutions to these requirements and extend available technology to seize the opportunities. Thus, technical and market research interact.

The development of training courses requires input from both marketing and technical research. However, courses are only part of the technology transfer process. Experts should present implementation workshops for the new technology. Consultants must be available to solve problems or, if a problem is not currently solvable, to stimulate research aimed at developing the technology needed for resolution.

The development of software tools should be seen as necessary to the application of a new technology and as an integral part of technology transfer.

It is useful for marketing personnel to attend training courses. They can facilitate connections between technical interests and project applications. This may involve uncovering and dealing with technological and cultural barriers to change. Technical marketing personnel can open communication channels, proactively soliciting user feedback that technology transfer personnel can use to improve tools, courses, consulting, research, and the marketing process itself.

These activities require a range of skills difficult to find in one person. Hence, building a team of people whose skills complement each other is essential, as is building trust and good communication within the team. In the future, we expect corporations to create technology transfer organizations specifically to improve and speed up the process of adopting software engineering technologies. These organizations will probably include a diverse group of professionals: researchers, educators, software developers, consultants, and marketing personnel. They will most likely associate closely with a research organization, but also have access to a training organization. They will cultivate extensive networks among practitioners. Technical improvement will depend on much more than technical factors alone.

We have tried to show why and how the 1990s will be the quality era for software. We believe that increasingly intense international competition will make it essential to specify and attain quantitative product characteristics in software-based systems. This will drive software engineering to become a true engineering discipline.

Existing technologies will become either better focused (reduced-operation software), more disciplined (reuse, measurement approaches), or more practical (formal development methods), providing more effective models for software development. The new disciplines of cognitive psychology and software sociology will enrich software technology.

Process and product models and other forms of structured experience will

aid in the practical engineering of software. Feedback and learning through measurement based on these models will become fundamental. Software models will become corporate assets, used not only for improving quality but also for transferring technology. Companies will plan for efficient technology transfer.

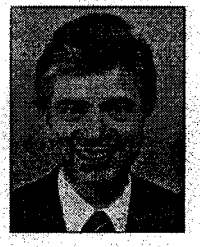
The justification for any measure is its role in helping satisfy user needs, and the importance of the measure is its correlation with this satisfaction. The presence of measures indicates that a technology is being challenged in a healthy fashion, that it is responding positively, and that it is therefore maturing. ■

Acknowledgments

We are indebted to Dieter Rombach, John Stampfel, Jar Wu, Pamela Zave, Marv Zelkowitz, and the reviewers for their helpful comments.

References

1. B. Boehm, *Software Eng. Economics*, Prentice Hall, Englewood Cliffs, N.J., 1981.
2. J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, New York, 1987.
3. J.M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, Vol. 23, No. 9, Sept. 1990, pp. 8-24.
4. B. Stroustrup, "What is Object-Oriented Programming?" *IEEE Software*, Vol. 5, No. 3, May 1988, pp. 10-20.
5. V.R. Basili, "Software Development: A Paradigm for the Future," *Proc. 13th Int'l Computer Software and Applications Conf.*, CS Press, Los Alamitos, Calif., Sept. 1989, pp. 471-485.
6. M. Kogure and K. Akao, "Quality Function Deployment and CWQC in Japan," *Quality Progress*, Vol. 16, Oct. 1983, pp. 25-29.
7. B.W. Boehm, J.R. Brown, and M. Lipow, "Quantitative Evaluation of Software Quality," *Proc. Second Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., Order No. 104 (microfiche only), 1976, pp. 592-605.
8. B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Comm. ACM*, Vol. 31, No. 11, pp. 1,268-1,287.
9. T.R. Riedl et al., "Application of a Knowledge Elicitation Method to Software Debugging Expertise," to be presented at the Fifth Conf. Software Eng. Education, Software Eng. Inst., Oct. 1991.
10. W.E. Deming, *Out of the Crisis*, MIT Center for Advanced Eng. Study, MIT Press, Cambridge, Mass., 1986.
11. *Transferring Software Eng. Tool Technology*, S. Przybylinski and P.J. Fowler, eds., IEEE CS Press, Los Alamitos, Calif., Catalog No. 887, 1988.



Victor R. Basili is a professor in the Institute for Advanced Computer Studies and the Computer Science Department at the University of Maryland. He chaired the department for six years. He is one of the founders and principals in the Software Engineering Laboratory, a joint venture between NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation. His research interests include measuring and evaluating software development in industrial and government settings.

Basili received a BS from Fordham University and an MS from Syracuse University, both in mathematics. He received a PhD in computer science from the University of Texas at Austin. He is currently the editor-in-chief of *IEEE Transactions on Software Engineering* and serves on the editorial board of the *Journal of Systems and Software*. He is treasurer of the Computing Research Board and an IEEE fellow. He is a former member of the Computer Society Board of Governors. He will chair the International Conference on Software Engineering in 1993.

John D. Musa is supervisor of the Software Reliability Engineering Group at AT&T Bell Laboratories. He has managed or participated in a number of software projects. His research interests include software reliability engineering, software quality, and software metrics.

Musa received a BA in engineering sciences and an MS in electrical engineering from Dartmouth College. He is a senior editor of the Software Engineering Institute book series; an IEEE fellow, elected on the basis of his extensive contributions to the field of software engineering and software reliability engineering over the past 15 years; a former chair of the IEEE Computer Society Technical Committee on Software Engineering and the Steering Committee of the International Conference on Software Engineering; and a founding member of the Editorial Board of *IEEE Software*.

Basili can be contacted at the Department of Computer Science, A.V. Williams Building, Rm. 411, University of Maryland, College Park, MD 20742; and Musa at AT&T Bell Laboratories, Rm. 2D248, 600 Mountain Ave., Murray Hill, NJ 07974.