Software Engineering in the Academy

Bertrand Meyer
Interactive Software Engineering

Institutions that teach software are responsible for producing professionals who will build and maintain systems to the satisfaction of their beneficiaries. This article presents some ideas on how best to honor this responsibility.

There is no universally accepted definition of software engineering. For some, software engineering is just a glorified name for programming. If you are a programmer, you might put "software engineer" on your business card but never "programmer." Others have higher expectations. A textbook definition of the term might read something like this: "the body of methods, tools, and techniques intended to produce quality software."

Rather than just emphasizing quality, we could distinguish software engineering from programming by its industrial nature, leading to another definition: "the development of possibly large systems intended for use in production environments, over a possibly long period, worked on by possibly many people, and possibly undergoing many changes," where "development" includes management, maintenance, validation, documentation, and so forth.

David Parnas, a pioneer in the field, emphasizes the "engineering" part and advocates[1] a software engineering education firmly rooted in traditional engineering—including courses on materials and the like—and split from computer science the way electrical engineering is separate from physics.

Because this article presents a broad perspective on software education, I will not settle on any of these definitions; rather, I would like to accept that they are all in some way valid and retain all the views of software they encompass. In fact, I am not just focusing on the "software engineering courses" traditionally offered in many universities but more generally on how to instill software engineering concerns into an entire software curriculum.

If not everyone agrees on the definition of the discipline, few question its importance. We might have wished for less embarrassing testimonials of our work's societal relevance than the Y2K scare, but it is still fresh enough in everyone's mind to remind us how much the world has come to rely on software systems. The institutions that teach software—either as part of computer science or in a specific software engineering program—are responsible for producing software professionals who will build and maintain these systems to the satisfaction of their beneficiaries.

SOFTWARE PROFESSIONALS

Judging by the employment situation, current and future graduates can be happy with their choice of studies. The Information Technology Association of America estimated in April 2000[2] that 850,000 IT jobs would go unfilled in the next 12 months. The dearth of qualified personnel is just as perceptible in Europe and Australia. Salaries are excellent. Project leaders wake up at night worrying about headhunters hiring away some of their best developers—or pondering the latest offers they received themselves.

0018-9162/01/$10.00 © 2001 IEEE

Although this trend shows no sign of abating in the near future, we should not take the situation for granted. An economic downturn can make employers more choosy. In addition, more people are learning how to do some programming, aided by the growing sophistication of development tools for the mass market. It is likely, for example, that many of the estimated six million people who are Visual Basic developers have not received a formal computer science education. This creates competition and will force the real professionals to stand out.

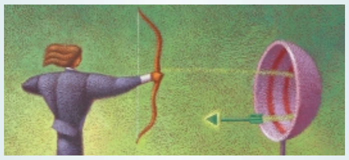
In fact, talking to managers in industry reveals that they are not just looking for employees—they are looking for excellent developers. This is the really scarce resource. The software engineering literature confirms[3] that ratios of 20 are not uncommon between the quality of the work of the best and worst developers in a project; managers and those who make hiring decisions soon learn to recognize where a candidate fits into this spectrum. The aim of a top educational program is to train people who will belong to the top tier.

Reflecting on why life has been so good, we may note that our constituency—the people who commission and use our systems—has been remarkably tolerant of our deficiencies. Users grumble about software, but get on with it. This may not last forever. Although relatively few people overall have been killed through the fault of software systems—a few dozen known cases so far,[4] a record that many well-established engineering disciplines might envy—a few high-profile cases would suffice to alert the public to the haphazardness of many of our methods.

Even if no such event occurs, the current problems with software have been serious enough to lead some government authorities, such as the state of Texas, to require licensing. Whatever you think of these initiatives (supported by Parnas's view that software engineers should indeed be licensed and registered), they reflect a general trend toward distinguishing the true software professional from the occasional programmer.

These trends have important consequences for universities. While just teaching use of the tools fashionable at a certain time may curry favor with students and their families (who can relate them to the skills most often requested in employment ads), doing so is not necessarily the best service you can give to future professionals. What matters is teaching them fundamental modes of thought that will accompany them throughout their careers and help them grow in this ever-changing field. The ones who blossom are those who can rise beyond the tools of the moment in harmony with the progress of the discipline.

This does not mean that we should neglect the tools of the trade. In any engineering discipline, these tools constitute a good part of the professional's bag of tricks. We must, however, look beyond them to the fundamental concepts, which have not changed significantly since they emerged when the field took shape some 30 years ago. As in hardware design, the technology evolves, but the concepts remain.



Although relatively few people have been killed through the fault of a software system, a few high-profile cases would suffice to alert the public to the haphazardness of many of our methods.

FIVE GOALS OF A CURRICULUM

A software curriculum should involve five complementary elements:

- *principles:* lasting concepts that underlie the whole field;
- *practices:* problem-solving techniques that good professionals apply consciously and regularly;
- *applications:* areas of expertise in which the principles and practices find their best expression;
- *tools:* state-of-the-art products that facilitate the application of these principles and practices; and
- *mathematics:* the formal basis that makes it possible to understand everything else.

Principles

Among the most important things that professional software engineers know are concepts that recur throughout their work. Most of these concepts are not specific techniques. If they include a technique, they go beyond it to encompass a mode of reasoning. This defines the most exciting aspect of being a professional software engineer: the mastery of some of these powerful and elegant intellectual schemes that, more than any particular trick of the trade, constitute our profession's common treasure. Most of them cannot be taught in one sitting but rather are learned little by little through trial, error, and skillful mentoring. I retain the same awe that I first felt when I started discovering them, and they make up the most important, if sometimes immaterial, body of knowledge that, as a teacher, I try to impart to novices.

The sidebar "The Principles: What Software Professionals Know" characterizes a few of these concepts. This list is not complete, but it should suffice to illustrate that much of what we have to teach we cannot just teach through words; it is a set of creative concepts that will come little by little as the amateur programmer progresses toward mastery of the trade.

Most of the people who are now in a position to influence a software curriculum started their careers when computers were an esoteric subject—just knowing how to use them set you apart from the rest of the population. Times have changed. Today, as Dennis

The Principles: What Software Professionals Know

Much of the body of software engineering knowledge consists of a set of recurring concepts that software professionals learn through trial, error, and skillful mentoring.

- *Abstraction*. This is our key intellectual tool—the ability to separate the essential from the auxiliary, to see the idea that presides over the realization and that it might initially hide.
- *Distinction between specification and implementation*. Getting this distinction right is a constant theme in software discussions:

  > "This is only an implementation detail!" "No, it's really part of what we want to do." "No, it's only one way of doing it!" "Show me another!" "It doesn't matter that I don't see another way at the moment; someone could come up with one later."

  This problem is unique to software because we deal with virtual, ethereal quantities. No one would confuse a bridge with the drawing of the bridge or a car with the plan of the car. You will not fall off the drawing into the water, and you cannot get run over by the plan. But in software, the distinction is often far from clear; unlike in Magritte's painting, we constantly risk confusing the pipe and the picture of the pipe. Learning to focus on the real issue is part of becoming a software professional.

- *Recursion*. The issue is not just recursive routines—a technique—but the general mode of reasoning that defines a concept by applying the definition itself to some of its parts. It is dizzying at first, but once you have learned to use recursion properly, you have gained a powerful intellectual tool, applicable throughout the field.
- *Information hiding*. Deciding what you export to the rest of the world and what you keep to yourself is a skill that software developers learn only through a combination of good examples and practice.
- *Reuse*. The good software developer soon realizes that one key to making your mark is to know when to rely on someone else's job. Reusing well is a skill; producing results for others to reuse is the sign of the master.
- *Battling complexity*. Software systems are ambitious intellectual undertakings—indeed, some of the most complex systems ever conceived by humankind—and complexity threatens to engulf us at every stage. The expert knows how to recognize the essential simplicity behind an apparent mess.
- *Scaling up*. Not only are some of our products complex, their sheer size can be staggering. The Windows 2000 source code takes up a reported 35 million source lines, and some defense or telecommunication systems are in the same range. A few million lines is common. Many software engineering issues take on a new urgency when size grows; the quantitative affects the qualitative. Part of a good professional's skill is knowing which techniques will scale up, if only because size is not always planned: Often, a large system is just a small system that grew.
- *Designing for change*. True to its name, software must change, can change, and does change more than engineering artifacts of any other kind. Unless developers thoroughly apply strict architectural principles, the change process can be painful, especially for large systems. Much of the justification for modern methods, languages, and tools is the expectation that they will facilitate this process. Even with systems of the size that a university education can tackle, we can let our students discover the challenge, learn the principles of designing for change, and directly experience the benefits.
- *Classification*. Object-oriented programming has shown that one way to attack complexity is to organize messes into smaller messes, and repeat the process.
- *Typing*. The notion that giving everything a type helps produce correct software elements, document them, and make them usable effectively is another realization that can deeply affect a software professional's grasp of the field—an observation that still applies to people who prefer untyped approaches for some of their work. Typing issues and techniques recur throughout the field, from specification to implementation and documentation. Our profession can boast of having taken the construction and study of type systems, object-oriented or not, further than any other profession; mastering their power is a required part of becoming a good software professional.
- *Contracts*. The practice of equipping algorithms, data structures, modules, and systems with precise constraints, guarantees, and invariants puts us in far better control of what we do. Once mastered, this skill will last a lifetime.
- *Exception handling*. When producing software, most of us would rather consider only the most desirable cases, but a professional must constantly worry about abnormal situations too. Only through systematic conceptual techniques can we avoid drowning the supposedly interesting parts of our reasoning—and of our programs—in myriad provisions for exceptional cases.
- *Errors and debugging*. Although textbooks usually do not emphasize this point, much of a software developer's daily life is spent dealing with things that don't work as they should, whether it's the developer's own fault or someone else's. Software professionals are the world's experts on messing up. We must teach students that this is a fact of life and show them how to deal with it.

Tsichritzis notes,[5] a software professional may well learn about key new developments from the morning paper. This is a humbling experience that challenges us even more to define, for ourselves and for our students, what entitles us to our claims of professionalism.

Practices

At a more mundane level, teaching software engineering also involves making the students familiar with practical techniques that have proved to be productive and are a key part of the trade. Examples include the following:

- *Configuration management.* Although it is one of the most important practices that every project should apply, configuration management is not used as widely or systematically as it should be. Configuration management is based on simple principles and supported by readily available tools.
- *Project management.* It does not always have to be such a hard task, but many software engineers are terrible at project management. Although the ample literature on software project management is not perfect, it contains gems that should be taught to all software students because most of them will at some point exert a project management role.
- *Metrics.* This is one of the most underused techniques in software development. Much of the current literature on metrics is not very good because it lacks a scientifically sound theory of what is being measured and why it is relevant. All the same, we should teach students how to use metrics to quantify applicable project and product attributes, to evaluate the claims of methods and tools through objective criteria, and to use quantitative tools as an aid to prediction and assessment.
- *Ergonomics and user interfaces.* Users of software systems expect high-quality user interfaces; like the rest of the system, the user interface must be engineered properly, a skill that can be learned.
- *Documentation.* Software engineers do not just produce software—they should also document it. A course on technical writing should be part of any software curriculum. Here engineering meets the humanities.
- *User interaction.* The best technology is useless unless it meets the needs of its intended users. A good software engineer must know how to listen to customers and users.
- *High-level system analysis.* To solve a problem through software, you must first understand and describe the problem. This task of analysis is an integral part of software engineering, and it's as difficult as anything else in it.
- *Debugging.* Errors and imperfections are an integral part of the software engineer's daily work. We need systematic and effective debugging techniques to cope with them.

It is not hard to find other examples of strong, robust techniques that every professional should know and practice.

Applications

Under the heading "applications," I include the traditional specific areas of software techniques: fundamental algorithms and data structures, compiler writing, operating systems, databases, artificial intelligence techniques, and numerical computing.

The aim here is not to be imperialistic by attaching these disciplines artificially to software engineering. On the contrary, it is to insist that whatever their individual traditions, techniques, and results may be, these are software subjects, and we should teach them in a way that is compatible with the particular view of software engineering the institution chooses. The advantage is mutual: The specialized subjects benefit from more methodologically aware students—for example, programming projects can focus on the subject at hand, rather than being distorted by pure programming issues because the students have already learned general design and programming skills—and they help meet software engineering goals by providing a wealth of new examples and applications.

Tools

The fashionable tools of the moment should not determine pedagogy. Indeed, Parnas has some rather strong words to say against teaching specific languages and tools. But if these aspects should not be at the center, we also should not ignore or neglect them. We must expose students to some of the state-of-the-art tools that industry uses. This exposure should proceed with a critical spirit, encouraging students to see the benefits and limitations of these tools—and to think of better solutions.

A tools curriculum cannot and should not be exhaustive; it is better to select a handful of programming languages and a few popular products and help the students understand them in depth. If they need other tools, they will learn them on the job. But they must have seen a few during their studies to have a general idea of what's available and what their future employers expect.

Inevitably, some skills become obsolete by the time a student graduates, but that is not such a bad risk

Teaching software engineering also involves making the students familiar with practical techniques that have proved to be productive and are a key part of the trade.

A sound educational program must resist the increasing attempts to impose specific tools on the basis of what's hot at the moment.

as long as the study of tools is a component of the curriculum, not its principal goal, and it is understood as the study of a few examples in light of more general principles. When the time comes to learn a more modern tool, having mastered some of its predecessors and understood their limitations is often helpful.

These observations apply in particular to programming languages. A strong software program will often choose its solutions in this area, a decision brilliantly pioneered when many departments chose Pascal in the 1970s. If the retained approach is not one of the dominant industry languages, the curriculum should also include a few service courses (or course sections) on these industry standards, both to let students have the right buzzwords on their résumés and to expose them to the variety of practical approaches popular in industry. A good software engineer is multilingual anyway, so there is no contradiction between using your approach of choice for teaching and letting your students discover as many others as time permits.

A sound educational program must resist the increasing attempts by interested parties—students' families are often among the most vociferous—to impose specific tools, in particular programming languages, on the basis of an assessment of what is hot in the employment ads of the moment. With the increasingly user-driven discourse of many universities, such pressures can be dangerously effective.

This is a recent trend. Had it been present 25 years ago, universities would never have moved to Pascal, as there were certainly no ads for Pascal programmers back then; industry, for the most part, had not even heard about Pascal. This phenomenon is of growing concern to many professors, who know that today's "in" skill may be tomorrow's dead end. While in a democracy we must take everyone's concerns into account, educators are responsible for choosing the appropriate tools on the basis of their best professional assessment of students' interests, not just in the short term but over the course of a career.

Mathematics

One example David Parnas gives of topics that he sees as belonging in a computer science curriculum—but not in a software engineering program—is denotational semantics. My view on this comes from my experience as a designer and software project manager in industry. Like Parnas, I am not that interested in students knowing arcane details of language theory at the PhD level, but I have come to treasure programmers' ability to reason formally. Although no ordinary software project applies formal semantics on a daily basis, some understanding that programming and programming languages are mathematical beasts susceptible to formal description is a key advantage, and I have found it to be particularly useful in software teams.

The level of such knowledge does not have to be very high. In *Introduction to the Theory of Programming Languages*,[6] I tried to summarize, in practical terms understandable by an ordinary programmer, the kind of formal semantic background that I would like a team member to have. Over and again in my development work, I have found that students who have mastered the ability to apply mathematical reasoning to software development have a distinct advantage over those who do not. In my opinion and experience, being a fully realized professional developer requires having mastered such topics as Hoare semantics and the basic techniques—such as present in *Z* or *B*—for modeling software issues in mathematical terms.

These skills are part of a strong university curriculum, and it's very hard to teach them through industrial courses after students have been immersed in production for too long. Referring again to Parnas's strong advocacy of the engineering side of software engineering, from my practical perspective as a producer of large systems for industrial customers, such abilities are more important for a team member than knowledge of materials engineering. One of the distinctive properties of traditional engineering disciplines is, after all, that they have a strong mathematical basis. Hoare semantics and the like are software engineering's closest counterpart to Maxwell's equations or the laws of mechanics.

The mathematics a software curriculum teaches should include the usual aspects—some calculus and discrete mathematics à la Knuth—but it also should provide a strong grounding in logic, an introduction to formal semantics in very practical terms, and some practice with a modern formal language and proof framework such as *Z* or *B*. Certainly, some students will at first resent being subjected to such material, but we can hope that many will become grateful, as they mature in their careers, that they were exposed to it early enough.

TEACHING BY DOING

Besides formal courses, any curriculum will include software projects. Often, they are insufficient preparation for the true challenges of professional software development, which involve large systems (rather than the relatively small endeavors of academic projects), large groups (seldom the case in a university setting), modifying someone else's legacy system (rather than building a solution from scratch), and dealing with end users (rather than a professor and a grader).

It is impossible, and may not be desirable, for a university setting to mimic these circumstances completely. After all, a university is not a company, and it shouldn't be. But we must prepare our students for the real challenges they will face. The standard academic project is not enough for this goal. An essential technique is the long-term project, which students should develop over more than a standard quarter or semester—typically over the course of a year. It should be a group project that includes aspects of analysis, design, and implementation. And it should involve the reuse, understanding, modification, and extension of existing software. The best way to achieve this last goal is to imagine the project running over several years, with each new class taking over the result of the preceding one and developing it further.

Such an endeavor may at first appear unrealistic, but a group of enthusiastic teachers at Monash University, under the direction of Christine Mingins, has been doing exactly that over the past few years. The result is an impressive graphical simulation program that every incoming class enhances, learning in the process the challenges and techniques of updating someone else's code.[7]

Obstacles exist, such as proper grading of such a project and the difficulty of organizing yearlong projects in a standard academic calendar. But with resolve and enthusiasm, we can address them. The result, I think, makes it worth trying.

THE INVERTED CURRICULUM

An idea that complements the multiyear project is to capitalize on one of the great promises of modern software technology: reuse. The principle of the inverted curriculum (a term borrowed from debates on electrical engineering education[8]), or "progressive opening of the black boxes" (a somewhat longer name but more precise[9]), is that the students first use powerful tools and components as clients for their own applications, and then progressively lift the hood to see how things are made, make a few modifications, and add their own extensions. The progression is from the consumer side to the producer side, but focuses from the start on powerful and possibly large examples.

There are several benefits. Right from the beginning, the students get to deal with impressive programs, like those that handle graphics. The teaching capitalizes on this "wow effect" and the ability to work with immediately visible results. Today's students have used electronic games and PCs from an early age, and they will not be too impressed by the typical introductory programming examples (the eight queens and such). Trying to get them excited is pedagogy, not demagogy.

Furthermore, the students learn hands on the necessity and benefits of abstraction and information hiding. We can try to teach these concepts abstractly through what will often sound to the students like homilies, but the laws of that genre inherently limit the effect. To realize that such discourse is not just an adult's injunction to be good but describes skills essential to survival in a large project, nothing beats having tried to use or modify a component and finding that it lacks proper specification separate from its implementation, proper contracts, and proper description of what should happen in exceptional cases.

I have discussed this strategy further in articles and a book chapter. I think it can be turned into a central feature of the curriculum. As far as I know, it has not been tried on a large scale yet, so it entails some risk and requires very careful planning, organization, and execution; but it is not out of reach for an ambitious and innovative institution. And it can dramatically alter the quality of the student experience.

Today's students will not be too impressed with typical introductory programming examples.

NONSOFTWARE STUDENTS

Besides teaching their own students, computing science departments are often asked to provide software courses for students majoring in other disciplines, especially branches of engineering. They may feel a tension between two trends: developing mini-CS curricula emphasizing the principles of the discipline itself, or treating such contributions as "service courses" that fulfill the exact needs of the customer, often focused on specific skills, languages, and tools.

This is a delicate issue, and it is easy to cite arguments for both sides. In support of the first, it is natural that computer scientists would want to focus on the contributions of their own discipline and try to convey some of its "dozen or two" foundational concepts. They may resent attempts by colleagues from other fields to interfere or be offended by an emphasis on purely utilitarian skills that downplays the intellectual contributions of their own field.

When programming education was introduced a few years ago into the curriculum of preparatory classes for the Grandes Écoles (France's elite engineering school system), official instructions directed teachers to use Pascal but stop at the introduction of recursion. What made this injunction particularly interesting is that the rest of the curriculum is heavily mathematical, involving quite abstract required subjects such as topology. Yet the powers in charge decided that recursion did not fit, as if recoiling at the potential effect on society of letting a few 17-year-olds

The industry claims wide use of reusable components, but has no quality guarantee, no standards, and no qualifying agency.

catch a glimpse of the infinite. They seemed to be sending a clear message: "You computer people don't have a discipline—you just know a few tricks that we want you to teach our students as they prepare to engage in serious intellectual endeavors." It is understandable that not all computer scientists would agree.

It would be unfair, however, to deny our client disciplines—the professions who will hire our students—their right to expect practical skills.

The solution is probably in-between: fulfill customer needs while controlling the pedagogy, and teach concepts as well as skills. As the discipline matures and loses its inferiority complex, it may find the task easier. Mathematicians too provide courses to other departments, but they seem to be able to teach mathematics the way they want while catering to their market.

A DEVELOPMENT PLAN

When considering the evolution of the field over the past decades, we cannot escape the possibly unpleasant observation that if much of the innovation in the 1960s and 1970s came from academic institutions, contributions from small entrepreneurial companies and the research labs of large corporations dominated the 1980s and 1990s. This is an overgeneralization, admitting exceptions, and may offend some readers. It would be hard, however, to point to many recent incontrovertible equivalents of such widely successful academic achievements as Wirth's Pascal, Dijkstra's THE operating system, Hoare's monitors and communicating sequential processes, the University of Oslo's Simula, and other milestones of the early years that showed universities could deliver not just good theory but also influential systems.

Part of the reason for this change is that the products of industry have raised the stakes. Wirth's compilers made the Swiss Federal Institute of Technology in Zurich a household name in the software world, but it is hard to imagine a compiler, however innovative, achieving a similar result today. A university group would have a tough time competing with the hundreds of developers behind Microsoft's Visual C++ or even those behind the GNU GCC compiler—not a commercial effort but also not an academic one in any accurate sense of the term.

People, students included, expect a compiler to come with a sophisticated development environment with all the trappings—a visual debugger, browser, graphical user interface designer, and configuration management facilities. For all the criticism that academic circles give Visual C++ and similar tools, which they may deserve in part, they provide a wealth of resources and facilities (some, it must be said, very cleverly devised), setting a high standard for anyone who wants to compete. If such competition is hard to sustain nowadays in former areas of academic excellence, academics must find new markets in which they can make their mark.

I make no pretense of knowing what all these new fields will be, but one that I find promising is the convergence of component-based development and quality. The industry claims that it is widely embracing the notion of reusable components. But there is no guarantee of quality for these components, no standard, no rules, and no qualifying agency. The risks are as huge as the opportunities. A major endeavor can fail in a catastrophic way because of a small deficiency in one of its more humble components. This kind of situation led to the failure of the initial launch of the Ariane 5 rocket—due to the poorly executed reuse of a minor software component—and delayed the entire industrial enterprise by a year and a half, costing European taxpayers an estimated $10 billion.[10]

Quality, however, is—or should be—academia's specialty. Huge opportunities can spring from this convergence. A long-term project, the source of PhDs, papers, industry collaborations, and a robust reputation, might involve

- defining standards for components;
- developing model high-quality components for everyone to appreciate, criticize, and emulate;
- setting up qualification metrics, possibly a component maturity model;
- setting up qualification suites;
- developing new methods and tools for better components, including proof technology, testing techniques, documentation techniques, and validation techniques; and
- setting up an organization to qualify and label components that third parties submit.

This kind of endeavor—although, of course, not exclusive of others in software engineering—can energize an ambitious computer science department and help it achieve an unchallenged level of international reputation, especially if complemented by an educational curriculum focused on software excellence.

For all the talk about "software engineering" in the literature, this article included, we must accept that the term remains in part a slogan, as it was when first introduced almost 35 years ago. Since then, however, we have learned enough to teach our students a coherent set of principles and techniques, without hiding from them—or ourselves—the many remaining uncertainties.

I have tried to maintain a balance here between the conceptual and the operational—the principles *and* techniques—as I think a software curriculum should do. I have tried to show that we do not need to sacrifice either of these aspects for the other, and to describe a challenge worth tackling: to set up a program of teaching and research that is at the same time serious, ambitious, attractive to the students, technically up to date, firmly rooted in the field's practice, and scientifically exciting. ✷

Acknowledgments

I thank C.A.R. Hoare and David L. Parnas for important comments on an earlier version of this article.

References

1. D.L. Parnas, "Software Engineering Programmes Are Not Computer Science Programmes," *CRL Report 361*, Communication Research Laboratory, McMaster Univ., Apr. 1998; to be published in *Annals of Software Eng.*, 2001.
2. Information Technology of America, "Major New Study Finds Enormous Demand for IT Workers: Research Pinpoints Hot Jobs and Skills Needed, Offers Insights on Employer Preferred Training Approaches," http://www.itaa.org/news/pr/PressRelease.cfm?ReleaseID=955379119.
3. B.W. Boehm, *Software Engineering Economics*, Prentice Hall, Upper Saddle River, N.J., 1981.
4. P.G. Neumann, "Illustrative Risks to the Public in the Use of Computer Systems and Related Technology," http://www.csl.sri.com/users/neumann/illustrative.html.
5. D. Tsichritzis, "The Changing Art of Computer Science Research," in *Electronic Commerce Objects*, D. Tsichritzis, ed., Centre Universitaire d'Informatique, Université de Genève, 1998.
6. B. Meyer, *Introduction to the Theory of Programming Languages*, Prentice Hall, Upper Saddle River, N.J., 1990.
7. C. Mingins et al., "How We Teach Software Engineering," *J. Object-Oriented Programming*, Feb. 1999, pp. 64-75.
8. B. Cohen, "The Education of the Information Systems Engineer," *Electronics & Power*, Mar. 1987, pp. 203-205.
9. B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, Upper Saddle River, N.J., 1997.
10. J. Jézéquel and B. Meyer, "Design by Contract: The Lessons of Ariane," *Computer*, Jan. 1997, pp. 129-130.

***Bertrand Meyer** is chief technology officer of Interactive Software Engineering, Santa Barbara, Calif., and an adjunct professor at Monash University, Melbourne. His books include* Object-Oriented Software Construction *(Prentice Hall, Upper Saddle River, N.J., 1997). Contact him at Bertrand_Meyer@eiffel.com.*