The international computer industry is perhaps the most competitive business of modern times, and the software component of that industry is rapidly becoming the driving force behind innovation in both software and hardware, as well as behind the resulting applications of computing. Trends into the early 1990s clearly point to software as a key enabling technology.

Yet, more than 20 years have passed while software development methodology has evolved to little more than a black art. The box box on p. 10 shows the evolution.

Why such slow progress in advancing human understanding of development processes, tools, techniques, and artifacts? Why has this key technology lagged behind other technologies of computing? Can anything be done to accelerate widespread advancement of software development technology?

We in the software industry are generally discontent with our poor understanding of the development process, the lack of tools for decreasing the time to market, and growing costs associated with development. These uncomfortable increases in time and cost are traced to the rising number and size of applications, complexity, and user expectations. These general factors are in turn related to the rise of graphical user interfaces, need for greater portability across platforms, trend toward networking of multivendor hardware and software systems, complex transaction-based distributed computing, market pull for more user customization, smart applications, and generally more demanding applications.

These topics recur in conversations with researchers and practitioners alike when discussing the state of software development. And these topics prompted this special issue on challenges of the 1990s. To gain a better appreciation for these challenges, IEEE Software organized a focus group made up of academic and industrial people and asked them to express their various points of view. After several meetings, this group quickly identified two broad themes:

- There is an untapped potential for productivity gains through the reuse of standard software components.
- There is a trend toward greater reliance on tools like rapid application development and design tools.

This special theme issue reports the points of view of several highly respected experts in various fields of development and software-engineering methodology. I believe you will be highly intrigued by the diversity of opinion represented by this
Software engineering is a systematic approach to the development, operation, maintenance, and retirement of software; software development is "the process by which user needs are translated into software requirements, requirements are transformed into design, design is implemented in code, and code is tested, documented, and certified for use," according to the IEEE Glossary of Software-Engineering Terms. Software engineering and software development have a long history of evolution:

- Before 1969: Software development is out of control because of cost overruns and failures, especially in operating systems development. The term "software engineering" is coined as the theme of the NATO-sponsored meetings in 1968 and 1969.
- 1969-1971: First principles are established through research into "good" programming practices. Advantages of top-down design, stepwise refinement, and modularity are recognized. New languages, including Pascal, and new group techniques, including chief programmer teams, are introduced.
- 1972-1973: Structured programming and notions of programming style emerge. The goto controversy subsides. Awareness of total software life cycle grows, and management and development aids are proposed.
- 1974-1975: Reliability and quality-assurance concerns give rise to systematic testing procedures, notions of formal program correctness, and models of fault tolerance and total system reliability. Early analysis of actual allocation of development effort and expense appears.
- 1976-1977: Requirements, specification, and design are the focus. Renewed attention is paid to early development phases before coding. Abstraction and modular decomposition are viewed as design techniques; structure charts and metacode are viewed as design representations. There are increasing efforts to integrate and validate successive development phases of the life cycle.
- 1978-1980: Dispersion and assimilation. The use of automated development tools increases: software-engineering courses are developed. The first principles of the 1969-1971 era begin to find widespread use in software industry.
- 1980-1989: This era marks the rise of CASE and the software-engineering workstation. Automated tools corresponding to each phase of the life cycle begin to appear on stand-alone workstations.
- 1990 and beyond: This era will see the application of expert-systems techniques to software engineering. The combination of software-engineering workstation, expert systems, and automated techniques for development will find widespread use in the software industry.

Greater standards efforts and the need for software developers to amortize the large development costs across many vendor's architectures are motivating portability at all levels — from source code to distribution-format conformity.

- Islands of computing. Computing has changed from a sole-source mode to a multivendor mode. In addition, the multi-sourced hardware and software must live in a networked world where programs and data can be interchanged and mixed at the user's request. This leads to the next two major trends:
  - Distributed, transaction-based computing. The network is rapidly replacing the mainframe as the backbone of corporate computing. In this new environment, micros, minis, and supercomputers all share a common interface via LAN and WAN connections. Lengthy batch computing periods are being replaced by short bursts or transactions, typically of a database nature.
  - Security and privacy. Interconnectedness of multivendor systems leads to the potential for insecure systems. A greater awareness of security issues is growing among system developers.

- Multimedia/image processing. The right brain is being challenged by greater use of graphics, complex image processing, and other forms of signal processing, including speech, music, video, and animation.

- User programming. The successful Hypercard product from Apple Computer, and similar user and customizable application software, has made the software industry keenly aware of this important segment of the user community. Some would say that the distinction between user and programmer is becoming unclear.

- Smart systems. Applications are being delivered with greater built-in expertise. These systems are characterized by intuition or heuristic knowledge.

- Interoperability. Developers are taking advantage of the Lego approach offered by off-the-shelf components that can be reused to form major parts of a new system without programming. A major theme just beyond the horizon is the concept of software-component manufacturers who produce reusable components for a wide variety of standard processing functions. Soon, developers will have a choice between buying or building.

- Development environments. Completely integrated development support

Continued on p. 12
environments are on the rise as part of the reusable-component movement. CASE vendors, for example, are rushing to standardize the interfaces to various tools and to cover the entire development life cycle. This has had side effects like earlier verification and more formal testing, change in how software is designed, increased use of specialized tools, requirements tracing, and need for more training.

- Object-oriented programming. Object-oriented design and object-oriented programming are rapidly moving into practice. This 20-year-old technology has finally come of age. Object-oriented features are being added to traditional languages like C, Pascal, and Fortran.
- Applications. Applications are broadening, becoming more embedded and transaction-based. And, as noted earlier, applications are becoming larger and more complex.

**Beyond the horizon**

If these are the trends, what should we do in the 1990s? This was the challenge to our distinguished authors, who were given extraordinary latitude in defining their own solutions to these problems. I believe this collection of articles makes extremely interesting reading because of their diversity of opinion, clarity of thought, and imposition on our sometimes preconceived ideas about development.

The first article, Mary Shaw's "Prospects for an Engineering Discipline of Software," compares software engineering with other engineering disciplines, like civil and chemical engineering. What can we learn from comparative study? Can software engineering benefit from lessons learned in other branches of engineering? What are some steps toward making software engineering a true engineering discipline?

Along the same line, but from a very different perspective, Brad Cox's "Planning the Software Industrial Revolution" traces the (American) history of invention and technology transfer from the 1700s to the present and then asks, "Can development be liberated from today's cottage-industry model by the application of interchangeable-parts technology?" Cox claims that nothing short of a revolution will do, and he outlines the steps in such a revolution.

John Musa and Bill Everett note the evolution of software engineering in "Software Reliability Engineering." They claim that we are in the last of four stages: functional, schedule, cost, and reliability. The reliability stage was prompted by the need to produce failure-free software to remain competitive. But, what do we mean by "failure-free"? Musa and Everett observe that measuring the number of failures per thousand CPU hours is more useful that the traditional count of failures per thousand source lines of code. How can we apply reliability analysis to practical software development? Read on!

Richard Cobb and Harlan Mills take a "trust, but verify" approach in "Engineering Software under Statistical Quality Control." Beginning with an analysis of common myths, they then provide very persuasive arguments for a statistical quality-control approach to development. Are software failures unavoidable? Does quality cost money? Is automatic machine verification impossible? Is software testing worthwhile? These and other provocative questions are answered in their article.

In "The Challenge of Building Process-Control Software," Nancy Lewson tells how real-time process-control development presents additional challenges beyond those of ordinary systems development. It introduces the notion of software safety, which is bound to play an important role in systems of the 1990s. What is software safety and how will it influence the 1990s?

"Iconic Programming: Where To Go?" by Tadao Ichikawa and Masahito Hirakawa describes the state of visual programming. This fascinating area of research holds great promise for user programming, but it is not without its critics. Is visual programming good for beginners but not for professional programmers? Is it restricted to small programs? Is visual programming a step forward, or merely a blind alley? I think you will be surprised at the answers the authors provide.

In "Joint Research between Industry and Academia," Carl Chang and George Trubow describe another challenge of the 1990s that has only been touched on by the previous authors. Technology transfer is vital to maintaining a competitive advantage, yet there have been very few successes. Is technology transfer feasible? How might we leverage software-engineering research by greater and faster transfer?

Finally, Tom DeMarco addresses perhaps the greatest challenge of all in "Making a Difference in the Schools." Development is an intensely intellectual activity, leading to the conclusion that we need highly educated software engineers. But are the schools providing this brain power? DeMarco lists several private-enterprise projects for improving education.

In the final analysis, the experts see: to confirm the focus group's conclusions that the challenge of the 1990s is to understand the process of development, automate the process as much as we can, and transfer innovation as rapidly as possible to remain competitive. For the details, I highly recommend that you read this issue from cover to cover.

**Ted G. Lewis** is a professor of computer science at Oregon State University, research director of the Oregon Advanced Computing Institute, and editor-in-chief of IEEE Software. His research interests include rapid application-development tools, parallel programming, and graphical-user-interface-management tools.

Lewis received a PhD in computer science from Washington State University. He is a member and Board of Governors member of the IEEE Computer Society and editorial board member of IEEE Spectrum.

**Paul W. Oman** is an assistant professor of computer science at the University of Idaho. Previously, he served as systems analyst and computer center director at various locations.

Oman received a PhD in computer science from Oregon State University. He is a member of ACM, IEEE, IEEE Computer Society, and Phi Kappa Phi. He is an associate editor-in-chief of IEEE Software, and the author of several articles on software tools and programming.

Address questions about this issue to Lewis at Computer Science Dept., Oregon State University, Corvallis, OR 97331-3902, Internet lewis@mist.cs.orst.edu.

IEEE Software