Where Is Software Headed?

To find out where software is headed, *Computer* took to the Internet, asking experts in academia and industry to share their vision of software's future. Their responses suggest a strong polarization within the software community.

The following roundtable of opinion is a sampling of the views of leaders in both academia and industry on the question of where software is headed. It is a snapshot in time of where we have been and possibly where we are headed.

This was supposed to be an introduction to the detailed comments on the following pages. After reading these selections, as well as others that were not chosen, I was struck by the chasm that exists between academia and industry. I had an epiphany, so to speak, and instead of my usual critical slam-dunking, came up with Table 1 which juxtaposes academic versus industrial world views. It appears that these two groups share radically different views on where software is headed. This difference may be more important than the individual items in the table.

The second impression, after realizing that the two groups are on different wavelengths, is the heavy emphasis on programming languages, operating systems, and algorithms by the academic group, in contrast to the clear emphasis on standards and market-leading trends by the industrial group. Academics worry about evolutionary or incremental changes to already poorly designed languages and systems, while industrialists race to keep up with revolutionary changes in everything. Academics are looking for better ideas, industrialists for better tools.

The final section in Table 1 may reveal the cause of this chasm. The academic group uses words like "efficiency, difficult problem, and evolution," while the industrial expert uses words like "time to market, opportunity, and revolution" to describe their world views. To an industrial person, things are moving fast—they are revolutionary. To an academic, things are moving too slowly, and in the wrong direction—they are only evolutionary changes which are slave to an installed base.

Whether you are in academia or in industry, I think you will find the following brief descriptions of where software is headed interesting and thought provoking. Let us know if you like this article or if you have other suggestions for how we can bring you up to date on the thinking of your peers.

—Ted Lewis, Naval Postgraduate School

Computer

Table 1. Academic vs. industrial world views.

Academic	Industrial
Parallel Processing	**Networked Processing**
• New portable languages (HPF)	• C++ for client/server
• Extensions to Unix	• Microsoft Windows, Novell, Unix
• Algorithms	• Rapid application development tools
OO Programming	**Object Technology**
• New languages (KidSim)	• OLE, OpenDoc, Smalltalk parts
• New operating systems	• DCE and/or CORBA
• Algorithms	• Business objects
Research	**R&D**
• New higher-level languages	• Productivity software
• Algorithms	• WWW tools
• Proof of Fermat's Theorem	• 3D graphics/GUIs
• Temporal databases	• Adding OT to RDBMS WWW search engines
• Isochronous network protocols	• Digital convergence
• Software agents	• Smart e-mail
• Multimedia software engineering	• End-user programming
• Formal methods	• Software tools
• Better undergraduate software engineering courses	• Improving group communications
Major Concerns	**Opportunities**
• Efficiency	• Time to market
• Difficult chronic problems	• Opportunity to make money
• Evolution	• Revolution

Dave Power, SunSoft
Bertrand Meyer, ISE Inc.
Jack Grimes and Mike Potel, Taligent
Ron Vetter, North Dakota State University
Phil Laplante, Burlington County College/
  New Jersey Institute of Technology
Wolfgang Pree and Gustav Pomberger,
  University of Linz
Mark Hill, James Larus, and David Wood,
  University of Wisconsin
Hesham El-Rewini, University of Nebraska
Bruce W. Weide, Ohio State University

The Desktop

# THE FUTURE OF SOFTWARE
Dave Power, *SunSoft*

The software industry is in the midst of a revolution, with developer limitations decreasing and user choice becoming increasingly a factor. The lower cost of network bandwidth and the emergence of object-oriented programming promise a bright future. Here are some of the things I see on the horizon: true networked computing, object stores, universal application access, ubiquitous information access, and the convergence of applications, content, and interactivity.

REALIZATION OF THE NETWORKED COMPUTING MODEL. There has been a lot of talk about client-server computing over the past few years, and it may seem fatuous to pitch it as a novel concept. But the reality of client-server is only now upon us. You can actually see this trend by looking at how big companies are positioning themselves. Sun has been saying for years that the network is the computer. Now Microsoft is also touting its networking capabilities, and IBM is calling itself network-centric.

True networked computing is just that: computing over a network, with applications, data, and processing power all dispersed across the network. Instead of focusing on individual CPU limitations, developers can write applications that take advantage of network bandwidth, make the most efficient use of the technology, and target the needs of the user. When companies begin to buy into a true networked computing model, software designers can begin to take full advantage of multithreaded, multiprocessing capabilities.

OBJECT-ORIENTED SOFTWARE. Today, we're already losing track of where application code lives. With object-oriented design, that code is moving with the object it describes, to enhance and manipulate it. The payoff with objects will come in application development. Developer productivity will be enhanced as application development becomes quicker. Soon there will be a standard way to define objects so they can be recognized and used by dif-

August 1995

21
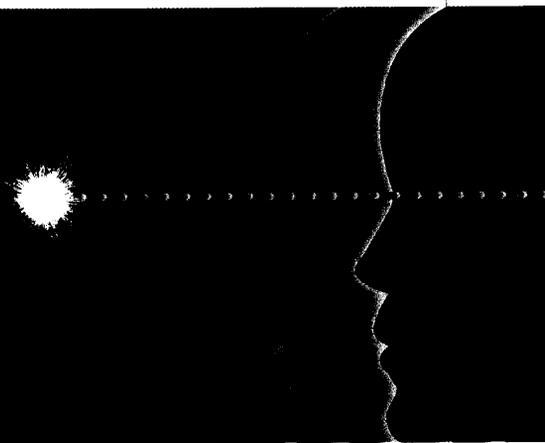
ferent operating systems, and stores of objects will reside on the network for different applications to use. Developers will be able to reuse or modify bits of applications—objects—that already exist.

Another benefit of object-oriented software is the creation of new business opportunities. In the future, application developers will specialize in horizontal and vertical object development, and we will see the emergence of specialized object developers in place of traditional independent software vendors. Financial instruments, for example, might be marketed as objects with performance histories and customization options for the consumer .

A UNIVERSAL DESKTOP? We won't converge upon a single operating system, simply because computing is not a one-size-fits-all venture. Different operating systems provide optimal environments for different types of customers. Unix, for example, provides strong support for networking and scalability, making it particularly well-suited for business enterprises. Windows and Macintosh offer a variety of very popular and versatile personal productivity applications, and personal digital assistants will eventually catch up to today's PCs. But application developers are using object technology to build portable applications that can run in any of these environments.

A world where one operating system reigns would be suboptimal: The operating system would be too big for some, too small for others. Like rosé wine, it would serve only people who can't make up their minds. Rather, integration should, and can, happen at a higher level than the operating system. The API (application programming interface) translation in such products as Wabi is one way to achieve this. The standardization of objects is another, as we've seen with object linking and embedding interoperability.

BUSINESS INFORMATION NAVIGATOR. Which brings us to the reason computers exist at all: information. There's a lot of hype about the information superhighway, and a lot of focus on the consumer market—making sure home consumers have access to on-line information, for instance. But what's true of the home market is even more true of the business market: Businesses need access to critical information; they need to know where to find it; and they need the tools to access it. The Business Information Navigator will emerge as the new "killer app."

Right now, it's easier to get information from a public-domain server in another country than to get a piece of information you might need from your company database. Imagine the equivalent Internet tools—ftp, telnet, gopher—put to work for internal information. Michael Crichton's *Disclosure* describes a visionary implementation in which users don a virtual reality helmet to access company information through a virtual interface that appears as an extended hallway of filing cabinets. Perhaps a more practical rendition of this idea is a workstation designed to let an office worker access information regardless of location—via the Internet or internally from corporate databases. A workstation that breaks protocol barriers to let knowledge workers—whether they be CEOs, marketing managers, or research assistants—get and use the required information. We're not talking about

increased CPU power; what we need are more search engines and protocol converters.

With the Business Information Navigator, it shouldn't matter what operating system you're running. Your desktop becomes more of an information access tool, and the integration comes from a higher level than the operating system.

CONVERGENCE OF APPLICATIONS, CONTENT, AND INTERACTIVITY. Again, there's a lot of talk about bringing interactivity to the home market. Almost all PCs built for home use have CD-ROM drives and multimedia capabilities. But the business market has more practical uses for audio, video, and interactivity capabilities, and more money to implement them. The products and services being brought to the home are, in fact, widely available in the business market but are grossly underused.

Text is limited. We achieve a greater ability to communicate by supplementing text with audio and video. Sun employees, for example, regularly receive both audio and video messages from Scott McNealy, Sun's CEO. For people at remote sites in particular, these messages are more personal and informative and more likely to be viewed or listened to than a text-only e-mail message. There is also a tremendous demand for videoconferencing. As the work force moves increasingly toward telecommuting, the ability to communicate visually from a computer will become even more critical.

To date, businesses really haven't taken advantage of the consumer trend toward new media. Companies are rallying to bring new delivery media such as cable and fiber optics to the home market, along with the receiving media (TVs and computers) to grab, store, and manipulate information. However, they are ignoring a potentially bigger user: the business market. At Sun, for instance, we have good, fast access to the Internet, from which users can access news from CNN or get real-time stock prices.

Many opportunities await us in the software industry, both in development of new applications and in new ways of doing business. We might also want to rethink the use of technologies already at hand. The time is ripe...let's seize the day. ❚

Dave Power is vice president and general manager at SunSoft PC Desktop Integration—a Sun Microsystems business unit. The SunSoft PC Desktop Integration unit develops and markets a suite of products that allow Solaris and other Unix users to run applications written for non-Unix environments. Power received his bachelor's and master's degrees in engineering from Tufts University and an MBA from Stanford University.

Readers can contact the author at SunSoft, Two Elizabeth Drive, Chelmsford, MA 01824; e-mail dave.power @east.sun.com.

FROM PROCESS TO PRODUCT: WHERE IS SOFTWARE HEADED?

Bertrand Meyer, *ISE Inc.*

To understand where software is going, we must realize that the evolution of software technology is not primarily determined by software technology. The idea that other factors also play a role is not original, but in most other advanced fields—think of computer hardware or genetic engineering—the primary factors are technological. If you study trends in electronic component integration and VLSI design, you have a good shot at finding out what computers will look like two, five, even ten years down the road. You must also consider market forces—will Intel hold its own against newly hungry rivals?—but the driving force is technology.

Not so in software today. Technology has taken a back seat to market considerations, largely because a single company, Microsoft, has been so incredibly successful at capturing not only market share but also mind share. No other major industry is so totally dominated by one player. This phenomenon is recent, and it is impossible to say how long it will last—that largely depends on how well or how poorly the Redmond team executes its next moves. But it will have as much influence on the evolution of software technology as anything that happens in a research lab, in a university classroom, or in the boardroom of another company. What is new is not that commercial factors are important for technology, but that they are so closely intertwined with technological factors and so often dominate them.

Here is a typical example. Software companies today are *removing* provably better features from their products because they do not "conform to Windows conventions." You get into an argument about why a product does something in a certain way; after a while everyone agrees that a certain user interface convention is good and that the reverse convention can cause trouble for users. And you comply, because if you do not follow the market you do not have a market. What is the lesson for the software technologist? *Not* that Windows conventions are all bad (many of them are excellent) but that in many cases being good or bad is less important than being the Windows convention.

That such events happen so commonly testifies to Microsoft's success and to its product quality; quarreling with this success would be futile and foolish. It would be just as absurd to deny the many positive effects that such standardization has had on a previously fragmented industry. And everyone knows that no empire is eternal.

Let's pretend for a moment, however, that we can ignore all this and concentrate on technology. Here is what I think—from an optimist's perspective—will change in software over the next few years. Reading the crystal ball is only fun if you make real predictions, so let's dive in:

- Reuse will become much more of a reality. The scene has already changed considerably. It's no longer necessary to preach reuse (although one does need to dispel reuse myths, which are gaining more ground as reuse progresses). Partly thanks to the level playing field established by Microsoft, we will see the current growing supply of reusable solutions turn into a real explosion.
- A reusability infrastructure will be built, based on the Internet. The problem of how to charge fairly for reusable software will be solved to the satisfaction of both producers and consumers.
- Object technology is here to stay. The real question is how long it will take for the general computing public to realize the limitations—already clear to most experts in the field—of first-generation hybrid approaches and adopt true object-oriented techniques, for all that they imply and, as a result, all that they bring.
- The "process culture" of traditional software engineering, which still dominates most software engineering literature and the major conferences, will at last yield to the "product culture" developed by the truly innovative and vibrant part of the industry—the people who make successful mass-market software for personal computers and workstations.
- I do not see much future in the next few years for some approaches that were recently heralded as promising: functional programming, logic programming, and expert systems (in their application to software). Some of them will find, or have already found, a niche, and all will remain useful as part of every software developer's bag of tricks, but it is hard to see how any of them could fundamentally affect our field over the next 10 years.
- Software education will improve, based on the increased understanding that there is a difference between knowing how to program a computer (increasingly a basic skill for the population at large, adding a P to the three Rs of K-12 education) and being a software professional.

The big question mark in the future is formal methods. It is difficult here, for someone like me who became a computer scientist by working on abstract data types and the original Z, to avoid mistaking wishful thinking for technology assessment. It is clear to all the best minds in the field that a more mathematical approach is needed for software to progress much. But this is not accepted by the profession at large. I can see two possible scenarios. The best one is that software science education will involve higher doses of formalism and that this will translate over time into a more formal approach in the industry at large. The other scenario is that formal techniques will be used only in high-risk development projects controlled by governmental regulatory agencies and will continue to exert some influence on the better programming languages. I must say that, wishful thinking aside, the last scenario is more likely—barring the unhappy prospect of a widely publicized, software-induced catastrophe.　　　　**I**

Bertrand Meyer *is with ISE Inc., Santa Barbara, Calif. He is an expert in object technology, the designer of the Eiffel language and associated basic reusable libraries, chair of the TOOLS conference, and the author of many books, including* Object Success: A Manager's Guide to Object Technology *(Prentice Hall, 1995). His e-mail address is bertrand@eiffel.com and his Web page is http://www.eiffel.com.*

SOFTWARE IS HEADED TOWARD OBJECT-ORIENTED COMPONENTS

Jack Grimes and Mike Potel, *Taligent*

Software development is becoming too expensive for the creation of high-function applications and systems. Further, these solutions must adapt to changing environmental conditions so that they can continue to meet their requirements. We believe that object-oriented, framework-based components are the preferred construction technology for developing software solutions that will be both flexible and economically constructed.

WHAT PROBLEMS DO COMPONENTS SOLVE? Large software systems have several problems that components, whether OO or not, can solve. One is the uncoupling of application and system development in both time (functional evolution) and space (geography). Separation in time means that an application or system can be released and years later, a component can be added to it and be functionally well integrated. Separation in space means that component development can be loosely coupled, so that developers need very little interaction or information about the internals of each other's software. What is necessary is agreement and standardization of the interfaces.

Another important problem is scale—the construction of applications and systems for complex problems. For example, one view is that C programs up to 50,000 lines can be successfully written using structured programming techniques. This is a crossover point where larger C programs must use object-based techniques (encapsulation of their data structures at all levels) to be successfully completed. "Successfully" emphasizes the delivery of software of sufficient quality to be used in mission- or business-critical applications. People will argue over the crossover point, which varies for each language, but most will accept the idea that the concept of encapsulation—for example, using abstract data types in Ada—becomes a necessity at some scale. By breaking larger problems down into many smaller ones, components help with the scale problem.

HOW DO COMPONENTS SOLVE THESE PROBLEMS? Applications and systems that support external components define an interface that is open (publicly available), so that developers can implement functionality independently from interface design. Good examples are the plug-ins available for the Adobe PhotoShop graphics package. Adobe provides the interface specification. Then, a hardware or software supplier can develop a product that can be accessed easily from within PhotoShop by the end user, as if the functionality were delivered "in the box" when it was actually developed and delivered independently of PhotoShop itself. More recently, the same approach has been taken by OpenDoc and OLE. The keys are the independence of development efforts, the potential separation in time of the development, and the factoring of problems into subproblems.

ARE OBJECTS NECESSARY TO BUILD COMPONENTS? No, but objects fit nicely since they provide fine granularity for hiding data structures. This is why there is the close associ-ation between objects and components. Using objects simply makes component interfaces easier to understand and components easier to create.

WHAT ARE THE ALTERNATIVES FOR COMPONENT CONSTRUCTION? Libraries, both procedural and object, are one alternative. They have existed for years and represent components in a primitive sense. The developer calls them, and they provide encapsulated functionality that can be reused. They can be developed independently in space and time, and provide a level of abstraction that helps with the problem of scale.

Object-oriented frameworks represent a more recent approach to component implementation. In addition to encapsulation, frameworks provide two additional benefits: flow of control and object orientation. Frameworks, as a grouping of classes that together provide a service, increase the level of abstraction. Frameworks also provide flow of control. This directly improves the scale of solutions that can be created because frameworks can be composed of other frameworks and represent the design of a service.

Object-orientation means more than encapsulation. For complex system construction from components, the level of abstraction provided by encapsulation is not enough. There is still a problem with the granularity of the components used. To simplify the development, certainly at the level of component assembly, one wants to use fewer, larger components. This fits well with the use of visual builders. However, to increase the generality of the solutions created, one wants more, smaller components. Larger components are less likely to deliver the needed functionality without customization. OO frameworks address both issues by allowing developers to modify existing components at various levels of granularity by providing two interfaces: an external, "calling" interface (like object libraries provide), and levels of internal, "be called" interfaces. Why should the granularity of functionality seen from the outside be the same as that available to the developer? The developer needs more flexibility than that provided by a "black box."

Frameworks provide this variable granularity in a way that doesn't compromise the developer's independence in time and space. A developer can deliver a smaller component that modifies the behavior of a previously delivered, larger component. OO frameworks provide this through inheritance and other OO techniques supported directly in C++, Smalltalk, Ada95, Eiffel, and so forth.

In the final analysis, the incorporation of object-oriented concepts of inheritance and polymorphism are an economic issue. That is, they improve the development costs and provide for reuse of design. Components implemented with object-oriented frameworks are simply a good solution to the problems of independence of development in time and space—and problems of scale.

WHAT ARE THE IMPLICATIONS OF COMPONENT SOFTWARE? A paradigm shift is occurring. Analogous to the paradigm shift in database technology 20 years ago when relational databases were introduced, the use of OO technology for component construction is in its early days. One of the implications for independent software vendors is the prospect of many more products, each smaller. While this will certainly occur in the longer term, many ISVs are using

Computer

component software mechanisms to implement next-generation, "full-sized" applications. They will deliver these applications to appear on the shelf, as they do now, with many components in the box. This provides the above-mentioned benefits to their internal development. In addition, it allows them to release modified, more specialized products for more narrow markets that weren't previously practical due to development cost.

For example, if a company wants a particular chart added to an ISV's charting application, this request will be much easier to accommodate if the developer can ship a component to the company that modifies the application to provide the required chart. Or, if the OO interfaces for the application are available to the in-house developers, they can make the change themselves. This incremental change is very hard to accomplish for a large application or component written in C, but may be quite easy for a component-based application written in C++.

ARE INFRASTRUCTURE CHANGES NECESSARY AS WELL?

Yes, long term. When component software is the norm, several changes will be needed. Software licensing will become more of an issue. Lots of components means that automated ways of tracking usage and royalties become more important. At some point, the cost of licensing a component must be less than the cost of a postal stamp.

For some combination of component size and communication bandwidth, electronic delivery becomes not only practical but required to match the delivery cost with the component cost. How will we pay for components delivered over networks? Fortunately, Visa and MasterCard are collaborating toward an answer.

Ultimately, we believe component software will fundamentally change the underlying programming systems used today. Systems such as Windows and Unix are procedural, library-oriented programming models designed to support the one-time development of monolithic applications of a certain size by a single programmer or team in one location. Object-oriented frameworks will facilitate the development of much more advanced and interoperable—but smaller— programs developed by multiple, independent programmers and teams. These programs will be customized repeatedly over time to meet changing needs.   ∎

Jack Grimes is director of technology evaluation at Taligent. Grimes earned a PhD in electrical engineering and computer science and two MS degrees, one in electrical engineering and one in experimental psychology. He has published on subjects from visual perception to VLSI graphics to object technology.

Mike Potel is vice president of technology development at Taligent. He received his BS in math from the University of Michigan and his MS and PhD in information sciences from the University of Chicago.

The authors can be reached at Taligent, 10201 N. De Anza Blvd., Cupertino, CA 95014-2233; email {jgrimes, potel}@taligent.com.

SOFTBOTS, KNOWBOTS, AND WHATNOTS

Ron Vetter, *North Dakota State University*

Distributed software agents (and network computing in general) will be the trend for software systems over the next five years. That is, we are headed for a computing environment where CPUs and storage devices will be widely distributed and connected by high-speed communication networks, making information readily accessible on a global scale.

Continued growth of the global information infrastructure, and its associated datasets, will cause information overload in every sense of the word. Simply put, the amount of information available in cyberspace is enormous and growing. Finding useful information, when it is needed, will be difficult. With an ever-expanding global Internet, it will become increasingly important to better understand how to design, build, and maintain distributed software systems.

One solution to this problem is the development of high-level software entities whose aim is to search for and find information of interest over this global network infrastructure. Several software systems exist today for this very purpose, and others are under development. These systems have a variety of names, including softbots, intelligent agents, knowbots, personal agents, and mobile agents. Since there are several interpretations for such systems, I will use the generic term *software agent* to mean a distributed computer program that is capable of carrying out a specialized function. In the context of this discussion, a distributed software agent is one whose goal is to intelligently find information of interest to users over a collection of heterogeneous networked computers.

What are the important issues for emerging software systems, given this anticipated trend in agent technology? How can the software community prepare for and contribute to this trend? How will different agents work together and how will they communicate? These are just a few of the questions that need to be addressed before agent technology becomes widely used.

First-generation software agents that reduce work and information overload have already been built and studied. Some of these agents provide personalized assistance with meeting scheduling, e-mail handling, electronic news filtering, and selection of entertainment. In each of these systems, agents are able to observe and imitate the user, receive positive and negative feedback from the user, receive explicit instructions from the user, and ask other agents for assistance when needed. These first-generation agents, though useful, still lack the structure needed to perform effectively in a large-scale global network environment.

Several algorithmic issues remain unresolved and need to be studied further by the software engineering community. For example, how will heterogeneous agents, built by different developers for different computing platforms, interact and collaborate. Consider the current state of affairs in distributed computing. Today, the client-server paradigm is the most widely used communication model for building distributed networked systems. Typically, in a client-server

paradigm remote procedure calls are used to facilitate client-server interactions over the network. In the future, a form of remote programming may well emerge, whereby the network carries objects (data and procedures) to be executed on remote machines. For example, General Magic has developed a software technology called Telescript that supports the development of distributed applications executing over a communication network. Telescript is an agent-based language that allows users to develop intelligent applications that are able to carry out specialized functions on behalf of the user.[1] Although this form of interaction is not yet well understood, it can simplify the development and introduction of new software systems.

One final issue is how agents will be programmed. That is, how will ordinary people tell agents what to do? This might be done using traditional programming languages, but not everyone knows (or wants to learn) a typical programming language. KidSim[2] offers one approach to this problem: It uses programming by demonstration and graphical rewrite rules to develop a system that allows children to program agents in the context of a simulated microworld. One of the principles found useful in developing such programming environments is to make the task visual, interactive, and modeless. Whether these ideas, or others, can be applied to software agents in general should be a fruitful area of research in the years to come. ∎

References

1. J.E. White, "Mobile Agents Make a Network an Open Platform for Third-Party Developers," *Computer*, Vol. 27, No. 11, Nov. 1994, 89-90.
2. D. Smith, A. Cypher, and J. Spohrer, "KIDSIM: Programming Agents Without a Programming Language," *Comm. ACM*, Vol. 37, No. 7, July 1994, 55-67.

Various pointers to information and resources concerning software agents can be found on the Internet at http://www.cs.umbc.edu/agents. Readers should also refer to the special issue on intelligent agents in *Communications of ACM*, Vol. 37, No. 7, July 1994.

Ronald J. Vetter is an assistant professor in the Department of Computer Science and Operations Research at North Dakota State University in Fargo. His research interests include high-performance computing and communications, multimedia systems, and high-speed optical networks.

Vetter received his BS and MS degrees in computer science from North Dakota State University in 1984 and 1986, respectively, and his PhD in computer science from the University of Minnesota, Minneapolis, in 1992. He is a member of the ACM and the IEEE Computer and Communications Societies.

Readers can contact the author at the Department of Computer Science , IACC Building, Room 258, North Dakota State University, Fargo, ND, 58105-5164; e-mail rvetter@plains.nodak.edu.

A RETROSPECTIVE LOOK FORWARD
Phil Laplante, *Burlington County College/ New Jersey Institute of Technology*

To understand where software is headed, it is interesting and informative to look back at where software has been. From its earliest inception as the reconfiguration of wires and switches, to machines codes, microprograms, and macroprograms, to assembly codes and higher order languages, the development of software has always been a quest for greater abstraction in support of greater complexity. And it is unlikely that software engineering will change the direction of that evolution.

As hardware systems become increasingly complex (billions of gates in newer systems as opposed to a few hundred in the first computers) and support more and different kinds of devices and applications, a richer framework for software engineering will be needed to permit the conversion of complex behavior from concept to a set of instructions that ultimately map into those gates. Fortunately, those very complex machines for which the software is targeted provide powerful platforms that can help us construct that software. For example, object-oriented methods require bulky compilers that generate relatively massive code. However, even the most modest personal computers are fast enough to mask the inefficiency of such primitive software engineering methods. I say primitive because although object-oriented techniques have been hailed as innovative and the solution to software engineering problems such as reusability, testability, maintainability, and so forth (the so-called "ilities"), it is deeply rooted in concepts that evolved in the 1970s with the revolutionary language CLU and in the theories of information hiding attributed to David Parnas.

Nor is any single framework for software engineering (object-oriented or otherwise) going to be sufficient. At a recent NATO Advanced Study Institute on Real-Time Systems, a distinguished panel was asked, "What is the proper software engineering framework for the development of real-time systems?" All six panel members answered differently. It is my steadfast opinion that development of a unified software engineering framework for all applications areas is folly. Efforts would be best expended concentrating on applications areas. Frankly, some academics (and some practitioners) have begun to wonder if computer science as a distinct field can survive; the study of architecture has become dominated by electrical engineers, the study of algorithms by mathematicians, and the study of software engineering by applications experts. Why shouldn't traditional software engineering be absorbed into applications disciplines? This eventuality represents somewhat of a full circle in the evolution of computing science—a science that was originally founded by physicists, mathematicians, and engineers.

To a certain extent I have avoided predicting the shape of software engineering in the next century. I have argued that as systems become more complex, so should software engineering techniques and tools. One prediction is that software engineering tools will expand beyond the rather flat, keyboard- and screen-driven interfaces. Human beings arguably perform more complex processing while

driving a car. Why not also have software engineering tools that involve interaction with sound and force feedback, and use of the eyes, feet, arms, voice, and head for input? The next software engineering tools will be more graphical, colorful, and musical. We must expand the portions of the brain used in creating computer programs; typing and clicking are not enough.

Current research focuses on artificially intelligent software systems using "intelligent agents" that can anticipate the needs of users and systems designers. While these agents are promising, we must never forget their origins in early expert systems (nested CASE statements) along with languages supporting data abstraction such as CLU (and successor object-oriented languages like Smalltalk). We must also never forget what software engineering tools are: abstraction mechanisms that simply represent layers of complexity that can now be smoothed over by increasingly fast hardware. The original Fortran compiler was introduced as an "automatic program generator"—the ultimate in artificially intelligent systems. (The Fortran compiler has not evolved much beyond its original, brilliant form.) New forms of "intelligent" software and software engineering tools will be largely incremental improvements on existing manifestations, with better interfaces enabled by faster hardware.

The sad fact is that software engineering has not evolved nearly as fast as the hardware has. And most of the innovations in software engineering and software systems are not profound innovations; rather, they are variations on very old themes. These innovations are primarily enabled by faster hardware and better interface devices. It is conceivable that the next generation of software will be just like this one. I don't mean to sound pessimistic or meanspirited—I am not. I simply am not allured by the bells and whistles attached to well-worn and well-known priniciples of software engineering. And I caution skepticism for some of the "snake-oil" that I feel is being sold as panacea.

What will the next generation of software be like? It will be more of the same, just bigger, faster, and with prettier wrapping.   I

Phil Laplante is dean of Burlington County College/ New Jersey Institute of Technology's Technology and Engineering Center in Mount Laurel, New Jersey. Prior to that, he was the chair of the Department of Computer Science and Mathematics at Fairleigh Dickinson University. His research areas are in software engineering, real-time processing, image processing, and real-time image processing. He also spent seven years in industry designing high-reliability avionics and support software. He is a licensed professional engineer in New Jersey and continues to consult to industry on real-time systems and real-time image processing. He is coeditor-in-chief of the journal, Real-Time Imaging.

Readers can contact the author at Burlington County College/New Jersey Institute of Technology, Technology and Engineering Center, Mt. Laurel, NJ 08054; e-mail laplante@njit.edu.

THE PAST AS PROLOGUE

Wolfgang Pree and Gustav Pomberger
University of Linz

Before we outline future software trends, let's briefly look back. The past has taught us that no single technology or concept constitutes a breakthrough. Computer-aided software engineering, prototyping, automated programming, object-orientation, and visual programming are just a few examples of technologies that have been heralded as a panacea for the known deficiencies of software development. But promising technologies are not applied immediately in industrial software development environments; indeed, it often takes decades for new technologies to have an impact outside of research laboratories. There are many reasons for this dilemma, the most important being that many companies are stuck with legacy software and often believe that they cannot afford to overcome this hurdle. The computer industry "helps" them by providing products compatible with the older ones.

Another phenomenon characterizes software development. Although the problems software attempts to solve are complex and thus pose difficulties in product development, unnecessary complexity is added in most software systems. Programmers are often proud of producing complicated solutions, and meticulous engineering is not rewarded.

What can we expect from the future? Extrapolating from the past, we envision a pessimistic scenario. The software crisis will grow ever worse with the addition of new domains and because new technologies and concepts won't migrate into the mainstream.

A look at current and soon-to-be-established de facto standards lends weight to such pessimism. Take object-oriented technology as an example. Though object-orientation could help overcome essential problems in software development, the most widely used object-oriented languages are antiquated; they are too complicated and thus provide no adequate tool for state-of-the-art software engineering. Unfortunately, higher level standards—for example, standards for object/component distribution and operating systems—are being built on top of these languages. Such premature standards add significant complexity to software products. Programmers are forced to produce unnecessarily complicated and unprofessional solutions for problems that could otherwise be solved much more efficiently.

Though standards are becoming the vogue in the computer industry, they perpetuate the software crisis. Despite much negative experience with de facto standards, the industry continues to adopt them. Thus, we predict that adopters of such standards will not be able to exploit the potential of the underlying concepts and probably will arrive at a dead end. It is simply too early to establish standards. Continuing to do so will create the impression—or reality—that marketing people and economic forces, not scientific advances, drive software technology.

The trend of forming increasingly larger project teams to develop software also exacerbates the software problem. Wirth states that "the belief that complex systems require armies of designers and programmers is wrong. A system that is not understood in its entirety, or at least to

a significant degree of detail by a single individual, should probably not be built."[1]

Unlike other engineered products, software is developed almost from scratch. The percentage of reused components is very low. Thus, software suffers from teething troubles and quality problems common to newly built products. This should not be necessary. Object-oriented concepts can overcome the reusability problem when they are used to build generic software architectures—that is, frameworks—for a particular domain so that components can be easily replaced or added. Again, existing and emerging de facto standards, as well as an industry that hesitates to apply this technology, delay the long-awaited breakthrough.

Though we draw a pessimistic picture of software's future, there is also hope. The future looks bright for those who depart from the well-worn path. An increasing number of companies that have applied computer technology almost since its inception, such as banks, recognize that they can no longer meet future requirements by merely maintaining legacy software. They have the chance to show courage by replacing the old systems with really new ones built without compromise.

Those who wait for a silver bullet will be disappointed. No single concept, method, or tool will result in a breakthrough. The key to successful software development lies in overcoming the obstacles sketched above and in applying a combination of already well-known concepts, methods, and tools to software development. We hope that the few who set off for new shores are so successful that the rest are forced to follow. ∎

References

1. N. Wirth, "A Plea for Lean Software," *Computer*, Vol. 28, No. 2, Feb. 1995, pp. 64-68.

Wolfgang Pree *is an associate professor of applied computer science at the Johannes Kepler University of Linz. He worked for many years in the area of object-oriented software development and has taught several courses at universities around the world, including Washington University in St. Louis, the University of Linz, and the University of Zurich. He is the author of* Design Patterns for Object-Oriented Software Development *(Addison-Wesley/ACM Press, 1995).*

Gustav Pomberger *is professor of computer science at the Johannes Kepler University of Linz. His research interests include prototyping, object-oriented software development, compositional software development, and software development environments. He is editor of the journal* Software—Concepts and Tools *and has published numerous articles and books in the field of software engineering.*

Readers can contact the authors at the C. Doppler Laboratory for Software Engineering, Johannes Kepler University, Altenbergerstrasse 69, A-4040, Linz/Auhof, Austria, {pree, pomberger}@swe.uni-linz.ac.at.

Parallel Software

PORTABLY SUPPORTING PARALLEL PROGRAMMING LANGUAGES

Mark D. Hill, James R. Larus, and David A. Wood
University of Wisconsin

Uniprocessor computers flourish while parallel computers languish. To a large measure, uniprocessors' success is due to a common and universally accepted programming model that has proven suitable for programs written in many styles and high-level languages. This model allows programmers to select the language most appropriate for expressing an application. Furthermore, programmers transfer most programs between computers without worrying about the underlying machine architecture (operating systems and user interfaces are, of course, another story). Computers did not always provide such a congenial environment. Several decades ago, every program was crafted for a particular machine in machine-specific assembly language.

Parallel computers still languish at this stage. They do not share a common programming model or support many vendor-independent languages. A program written for a workstation will not exploit the parallelism in a shared-memory multiprocessor. Similarly, when a program exceeds the resources of a bus-based multiprocessor, it must be rewritten for the message-passing world of workstation clusters or massively parallel processors.

High Performance Fortran (HPF) is a ray of light in this bleak world. Vendors across the entire spectrum of machines have announced HPF compilers. Unfortunately, HPF is a domain-specific language targeted at a narrow range of applications whose primary data structure is dense matrices. If your application can be written easily in Fortran 77, it may run in parallel in HPF.

General-purpose parallel languages cannot succeed without a common underlying model that gives programmers intuition as to the cost of operations and compiler writers a common basis for implementing these languages.

What is this common model? We believe it is a shared address space in which any processor can access any shared datum at a uniform, processor-independent address. A shared address space extends the uniprocessor model in a way that preserves programmers' expertise and supports the complex, pointer-rich data structures that underlie most large applications. Processor-independent addresses also allow dynamic load balancing and transparent data partitioning.

Note that a shared address space does not require shared-memory hardware. The latter is only one implementation technique. Languages such as HPF and runtime libraries such as the University of Maryland's CHAOS library for irregular applications implement a shared address space using compilers or runtime code. However, these languages and libraries are narrowly focused on particular application domains and do not support a common programming model. Other languages, such as Split-C, aim for a wider domain of applications, but their programming model remains closely tied to a particular type of machine.

To address this problem, the Wisconsin Wind Tunnel research project has developed the Tempest interface, which provides a common parallel computer programming model. Tempest consists of a substrate—implemented in either software or a combination of hardware and software—that allows compilers and programmers to exploit different programming styles across a wide range of parallel systems.

Tempest provides the mechanisms necessary for efficient communication and synchronization: active messages, bulk data transfer, virtual memory management, and fine-grain access control. The first two mechanisms are commonly used for short, low-overhead messages and efficient data transfer, respectively. The latter two mechanisms allow a program to control its memory so that it can implement a shared address space. Fine-grain access control is a novel mechanism that associates a tag with a small block of memory (for example, 32-128 bytes). The system checks this tag at each Load or Store. Invalid operations—loads of invalid blocks or stores to invalid or read-only blocks—transfer control to an application-supplied handler.

Because Tempest provides mechanisms, not policies, it supports many programming styles. Current parallel machines are designed for a single programming style—message passing or shared memory—which forces programmers to fit a program to a machine rather than allowing them to choose the tools appropriate for the task at hand. Programs written for a particular parallel machine are rarely portable, thus limiting the appeal and use of these machines. By separating mechanism from policy, Tempest allows a programmer to tune a program without restructuring it. In particular, Tempest allows a programmer to select (from a library) or develop a custom coherence protocol that provides an application with both a shared address space and efficient communication.

Tempest's success depends on effective implementations throughout the parallel machine pyramid. Symmetric multiprocessors (SMPs) form the base of this pyramid. Most programs are, and will continue to be, developed on these inexpensive and ubiquitous machines. Larger jobs with low communication requirements may require a step up to networks of desktop workstations (NOWs). Networks of dedicated workstations, possibly with additional special hardware, can trade higher cost for increased performance. Finally, at the pyramid's apex, supercomputers and massively parallel processors (MPPs) offer the highest performance for those able to pay for it.

We have developed several Tempest implementations. Typhoon is a proposed high-end design. It uses a network interface chip containing the interprocessor network interface, a processor to run access-fault handlers, and a reverse translation lookaside buffer to implement fine-grain access control. The Blizzard system implements Tempest on existing machines without additional hardware. It currently runs on a nonshared-memory Thinking Machines CM-5 and a network of Sun Sparcstations and uses one of two techniques to implement fine-grain access control. Blizzard-E uses virtual memory page protection and the memory system's ECC (error-correcting code) to detect access faults. Blizzard-S rewrites an executable program to add tests before shared-memory Load and Store instructions.

Preliminary performance numbers show that with adequate hardware support, shared memory implemented on Tempest is competitive with hardware shared memory. However, the real benefits and large performance improvements arise from the custom coherence protocols made possible by Tempest. As our experience with Tempest grows, we continue to refine it. Whether it becomes, or influences, a standard substrate for parallel computing remains to be seen. Nevertheless, we believe that a multiparadigm, portable, standard substrate is essential if parallel computers are ever to flourish. ∎

Mark D. Hill is an associate professor in both the Computer Sciences Department and the Electrical and Computer Engineering Department at the University of Wisconsin-Madison. His work targets the memory systems of shared-memory multiprocessors and high-performance uniprocessors.

He earned a BSE in computer engineering from the University of Michigan (1981) and an MS and PhD in computer science from the University of California-Berkeley (1983 and 1987). He is a 1989 recipient of the National Science Foundation's Presidential Young Investigator award, a Director of ACM SIGARCH, a senior member of IEEE, and a member of IEEE Computer Society and ACM.

James R. Larus is an associate professor in computer sciences at the University of Wisconsin-Madison. His research interests include parallel programming, programming languages, and compilers. He received his AB in applied mathematics from Harvard University (1980) and an MS (1982) and PhD (1989) in computer science from the University of California at Berkeley. He is a 1993 recipient of the National Science Foundations's National Young Investigator award and a member of IEEE Computer Society and ACM.

David A. Wood is an assistant professor in both the Computer Sciences Department and the Electrical and Computer Engineering Department at the University of Wisconsin-Madison. His interests include the design and evaluation of computer architectures, with an emphasis on memory systems for shared-memory multiprocessors. He received a BS and PhD in Computer Science from the University California-Berkeley (1981 and 1990). He is a 1991 recipient of the National Science Foundation's Presidential Young Investigator award and a member of ACM, IEEE, and the IEEE Computer Society.

The authors can be contacted at the Department of Computer Sciences, University of Wisconsin–Madison, 1210 West Dayton Street, Madison, WI 53706; e-mail {markhill, larus, david}@cs.wisc.edu; http://www.cs.wisc.edu/~wwt.

Parallel bits

Hesham El-Rewini, *University of Nebraska at Omaha*

arallel computing has experienced a number of setbacks over the past few years. Several parallel computer manufacturers went out of business, a number of parallel programming languages proved unsuccessful, and parallel applications development is still far from being an easy task. There is a recent wave of skepticism about the future of parallel computing. Will parallel computing continue along the same lines? Will it set off in new directions? Or will it quietly expire?

Parallel computing will survive by setting off in new directions. Traditionally, scientific computation has been the major driving force behind parallel computing. Today, commercial applications are emerging as another significant force driving the development of future parallel systems. In addition to computation-intensive scientific applications (such as numeric simulation of complex systems), data-intensive business applications (such as videoconferencing, advanced graphics, and virtual reallity) will begin to take advantage of parallelism.

The new advances in network technology have narrowed the distinction between the two worlds of parallel and distributed computing. It is now feasible to develop applications on remotely distributed computers as if they were parts of one parallel computer. This trend will continue to flourish in the future as the reliability of such systems improves. Future parallel systems will be networks of heterogeneous computers comprising some or all of the following: workstations, personal computers, shared-memory multiprocessors, and special-purpose machines. We will witness greater integration of parallel computation, high-performance networking, and mutimedia technologies. Naturally, this will influence the design of operating systems and programming languages.

In an attempt to gain some insight on the future of parallel computing, I asked several scholars active in this field to speculate about its future. Below, I set forth their thoughts as they relate specifically to the software aspects of parallel computing.

THE FUTURE OF PARALLEL COMPUTING

We will see general-purpose parallel computing within the next 10 years. Just as standard languages and portable software made sequential computers a universal form of computing, standard parallel languages must be developed if parallel computers are to achieve the same degree of popularity. Moreover, these languages (or software in general) will need to be portable, enabling the user to ignore the implementation details of a given platform.

Future parallel computer programmers will not concern themselves with the tedious task of how parallelism is achieved. Compilers or operating systems will take charge of distributing parallelism onto different processors and also of exploiting levels of parallelism in a particular program (or application). If parallel computing is to gain more acceptance, the programming of such computers should be made easier, even if we have to sacrifice some performance.
—*Albert Zomaya, The University of Western Australia*

APPLICATIONS

By sheer numbers, embedded systems, particularly in the area of signal processing, account for a large percentage of parallel applications. This technology is distinct from the "distributed" computing perspective.

The "application package" approach, which hides the implementation of parallelism, may force a programmer to take an inherently parallel problem and code it into a sequential solution. Parallelizers will never do well in finding and exploiting the original parallelism of such a solution. Areas such as signal or image processing are moving toward methods that let programmers express inherent parallelism while requiring little system overhead.

I will add my own opinion that the one common need across many different parallel system domains is for increased fault tolerance, both hardware and software supported.
—*Scott Cannon, Utah State University*

BUSINESS APPLICATIONS. Business applications will define the market for parallel systems applications, with numerically intensive industrial code running efficiently on parallel systems. Programming languages will be predominantly high-level application/field-specific or user-oriented (graphical) problem-specification languages that offer transparent exploitation of parallelism. Intelligent and fully integrated programming environments will become available, incorporating application/field-specific user interfaces and interactive programming support, as well as methods and techniques for efficient reuse of software at different levels of abstraction.
—*Karsten M. Decker, Swiss Scientific Computing Center*

ACADEMIC RESEARCH. In 10 years, 95 percent of the machines will be shared-memory multiprocessors that run databases like Oracle, Informix, and so on for business applications. The scientific market will distinguish itself only by adding large memories to Cray and SGI boxes. But this will be only about 2 percent of the market.

Researchers will continue to make slow progress toward useful tools—for example, languages, operating systems, and visualization tools; this will keep us busy and funded but out of the mainstream. Perhaps in 10 to 20 years some of this will pay off, and companies will actually use it!
—*Ted Lewis, Naval Postgraduate School*

GETTING SERIOUS. If parallel processing is to grow, it has to adapt to popular applications. We seem to be headed toward graphics-driven applications, mainly in the form of games. It is time for parallel processing to move from scientific applications to everyday applications in business and recreation. There is

potential for applying parallel processing to spreadsheet applications for modeling large systems.

—Ted Mims, University of Illinois at Springfield

LANGUAGES

A "functional" version of C and Fortran will prevail and facilitate parallelizing compilers. Parallel programmers will still have access to pointers and other imperative features of C/Fortran but in a limited way. Parallel versions of C/Fortran will not survive, since they are the equivalent of assembly programming in sequential programming. Tools will also play a very important role in developing parallel applications and making them portable across different architectures.

—Behrooz Shirazi, The University of Texas at Arlington

POLYGLOTISM. There will be lots of parallel programming languages, and most people will use an application package that hides the parallelism from them. It will just look like a very fast sequential computer.

—Michael J. Quinn, Oregon State University

A CAVEAT FROM THE COMMERCIAL WORLD

In the commercial world of Windows and Windows NT, it is hard for me to envision parallel programming ever becoming mainstream unless parallel programming (constructs, data, and so forth) is subsumed by the tools, language, and the underlying hardware. The average Windows programmer has a hard enough time getting multiple threads running right without getting deadlocked. Given this, parallel programming is either relegated to the dustbin or to the chosen or brave few. However, I do hope that operating systems use more PP constructs. Almost all OS houses now have 32-bit systems and support of multiple threads.

In 10 years, I envision machines in the commercial arena with dozens (but less than 100) of nodes and with shared memory. Shared memory challenges the average programmer who will never get the concept of each processor having its own memory right. Yes, an OS can provide a single "virtual memory" over a distributed-memory system. But today's hardware has performance limitations on how "smoothly" virtual memory can be mapped over distributed memory. In fact, database programmers have begun noticing bottlenecks on Sequent machines that have 4-16 nodes and multiple memory modules. Perhaps in 10 years the hardware limitations will be overcome so that we can have multiple nodes with independent memory virtualized as a shared-memory system by the OS.

I also envision coarser grained objects than we have today. Thus, one can imagine multiple objects working in parallel to solve a problem. These objects will in turn use the underlying PP constructs (but remember...the chosen few rule). I certainly hope that we get better language and tools support to bring PP to the average programmer.

—Alok Sinha, Microsoft

CHALLENGES OF SOFTWARE DESIGN AND THE UNDERGRADUATE COMPUTING CURRICULUM

Bruce W. Weide, *Ohio State University*

Almost from its inception, the software industry has endured a perennial state of crisis. We've all heard the complaints. "Studies have shown that...some three quarters of all large systems are 'operating failures' that either do not function as intended or are not used at all."[1]

Why is the crisis mentality—and the crisis itself—so prevalent, so persistent? Educators' contributions to the muddle should not be overlooked. As an academic computer scientist, my observations on the future of software address the centrality of software design in software engineering and the direction of software engineering education.

Design here includes not just the traditional high levels of user-interface design and macroarchitecture (boxes and arrows, flows, and so forth), but also microarchitecture (component interface design) and details right down to the code level. Poor design is a major culprit in the software crisis, and CS curricula should treat software design differently than they do now.

THE SOFTWARE DEVIL IS IN THE DETAILS. Many respected voices in computing call for a relatively quick and easy attack on the software crisis: greater emphasis on software engineering processes. The backers of this position generally claim that improved management is more important than improved technical solutions (the technical problems having already been "solved"). A serious problem with this approach is its underlying assumption that product quality derives largely from process quality, that high-quality products inevitably result when people are well managed. These are dubious propositions at best, especially for an emerging field such as software. There is just no solid evidence to support such wishful thinking. After all, if the code doesn't work or is unmaintainable as designed, nice high-level pictures and process certification won't help much.

Meanwhile, many other computer scientists believe that the best response to the software crisis lies in improving understanding of software itself—hence, the recent emphasis on formal methods, object-oriented design, component-based design, and so forth. One feature of this approach is that it is a long-term proposition, requiring additional fundamental research. Because of its inherently more technical nature, technology transfer in this approach relies largely on a "bottom-up" infusion of new ideas through the entry of recent graduates into the workforce.

I concede that process issues play a nontrivial role. But if software engineers in the trenches do not know how to design well, no amount of clever administration or management can produce high-quality systems. Other engineering disciplines acknowledge this. We should, too.

CURRENT CS CURRICULA DO NOT ADEQUATELY ADDRESS DESIGN. What are the implications of this centrality of design in software engineering? Most CS curricula

August 1995　　　31

include a sequence of courses emphasizing software design. Even assuming that the instructors of these courses teach—and students actually learn—what the instructors intend, graduates are unlikely to escape (much less help solve) software's chronic crisis. Unintegrated course sequences on software system design can have only limited impact on the practices of future software designers. Why? The instructor in the next course down the line has a completely different view of how software should be designed, or possibly no well thought out view at all. Students quickly fall out of practice in applying concepts and methodologies just learned, especially if they get the message (even implicitly) that those particular concepts and methodologies are not so important or fundamental or necessary.

Why doesn't every decent undergraduate computer science program advocate a specific, detailed approach to software design and development and teach it in depth? There are many problems, not the least of which is that it is as hard to teach good design as to do good design. This fact of life is a problem for traditional engineers. But software engineers face the added difficulty that most educators cannot even agree on what a well-designed software system should look like, and the same is true for practitioners. It is easy to observe this by examining computer science textbooks, technical papers, and commercial "industrial-strength" software. Except for egregiously poor design practices, most software engineers and software engineering educators cannot separate fair-to-good software designs from excellent ones. Beyond the tenets of structured programming, few accepted community standards stipulate what software systems should be like at the detail level.

This situation presents a clear problem for educators: Exactly what should we teach regarding design? But there is a less obvious problem, too. If the instructors in a course sequence do not reinforce one another's ideas about the details of how software should be designed, students get mixed signals and conclude that those details do not really matter, when precisely the opposite is true.

To avoid the limitations of single-course efforts and the mixed signals sent by an unintegrated course sequence, effective software design instruction demands a critical mass of faculty with a shared vision of how to design and develop industrial-strength software systems. These faculty must be involved in the entire design sequence starting from CS1. Perhaps at least four to five should be "on board"—that is, agree on the details of the approach and design principles and technology to be taught. I know from experience that assembling such a team is no small feat, because faculty members are usually militant individualists, especially when it comes to how software ought to be designed. But it is not impossible.

Readers interested in contributing a short article (1,000-1,500 words) to an upcoming roundtable on object technology should contact Scott Hamilton at s.hamilton@computer.org by August 21, 1995.

To the best of my knowledge, no existing curricula are comparable—in content or in level of integration—to what is required to address the educational needs of designers and developers of software systems. Developing a consensus on any detailed design approach that can be shown to lead to high-quality software systems and creating an integrated curriculum based on it are important challenges for software educators over the next decade. ∎

Reference

1. W. Gibbs, "Software's Chronic Crisis," *Scientific American*, Sept. 1994, pp. 86-95.

Bruce W. Weide *is an associate professor of computer and information science at Ohio State University in Columbus. His research interests include reusable software components and software engineering, including software design, formal specification and verification, data structures and algorithms, and programming language issues.*

He received a BSEE degree from the University of Toledo and a PhD in computer science from Carnegie Mellon University. He has been at Ohio State since 1978. He is codirector of the NSF- and ARPA-supported Reusable Software Research Group at OSU, which is responsible for the Resolve discipline and language for component-based software (see URL http://www.cis.ohio-state. edu/hypertext/rsrg/RSRG.html).

Readers can contact the author at the Department of Computer and Information Science, Ohio State University, 2015 Neil Avenue, Columbus, OH 43210; e-mail weide@cis.ohio-state.edu.