Gaining Intellectual Control of Software Development

Barry Boehm
USC Center for Software Engineering
Victor R. Basili
University of Maryland

The results from two workshops on software engineering research strategies, commissioned by the National Science Foundation last year, hint at new directions that software development might take.

Recent disruptions caused by several events have shown how thoroughly the world has come to depend on software. The rapid proliferation of the Melissa virus hinted at a dark side to the ubiquitous connectivity that supports the information-rich Internet and lets e-commerce thrive. Although the oft-predicted Y2K apocalypse failed to materialize, many software experts insist that disaster was averted only because countries around the globe spent billions to ensure their critical software would be Y2K-compliant. When denial-of-service attacks shut down some of the largest sites on the Web last February, the concerns caused by the disruptions spread far beyond the complaints of frustrated customers, affecting even the stock prices of the targeted sites.

Indeed, as software plays an ever-greater role in managing the daily functions of modern life, its economic importance becomes proportionately greater. It's no coincidence that technology stocks have led the upsurge of stock market indices, that the US government's antitrust case against Microsoft has become headline news around the world, or that some companies' aggressive pursuit of software patents has caused widespread controversy.

Yet despite its critical importance, software remains surprisingly fragile. Prone to unpredictable performance, dangerously open to malicious attack, and vulnerable to failure at implementation despite the most rigorous development processes, in many cases software has been assigned tasks beyond its maturity and reliability.

TAKING THE LONG VIEW

Fortunately, the groundwork for finding solutions to many of these shortcomings has already been laid. In 1997, recognizing the US economy's increasing dependence on information technology, the Clinton administration established the President's Information Technology Advisory Committee. PITAC's primary responsibility was to determine whether government-supported R&D "is helping to maintain United States' leadership in advanced computing and communications technologies and their applications." After many meetings with the research community, funding agencies, industry, and the public at large, PITAC issued a hard-hitting report that recommended increasing government-

You cannot go from good to great IT systems if you improve only the SE elements and neglect the IT components.

sponsored IT research by $1.37 billion annually within five years.

To support their findings, the committee found that the "Federal information technology R&D investment is inadequate ... [and] too heavily focused on near-term problems" and recommended "a strategic initiative in long-term information technology R&D," which should sponsor "research that is visionary and high-risk." The committee focused on software because the nation has "become dangerously dependent on large software systems whose behavior is not well understood and which often fail in unpredicted ways." Improving software development methods is so important that, the committee argued, we should "make software research a substantive component of every major IT research initiative."

In response to these recommendations, the computing directorate at the National Science Foundation commissioned two major software workshops last year.

NSF WORKSHOP FINDINGS

The first workshop, the NSF Workshop on a Software Research Program for the 21st Century, concluded that software research must expand the scientific and engineering basis for constructing no-surprise software of all types.[1] To do so, workshop participants found that software developers need to

- "Develop the empirical science underlying software as rapidly as possible. One important activity will be to analyze how some commercial and government organizations have learned to build no-surprise systems in stable environments. By extracting principles from these analyses, empirical research can help enlarge the no-surprise envelope. By validating principles derived from theoretical research, where many excellent but unused ideas originate, it can enlarge the toolkit of software developers.
- "Advance our understanding of the basic elements of the computer science discipline, which is the foundation for all software construction. Progress in formal methods, algorithms, operating systems, database management systems, programming languages, and many other areas is essential. Otherwise, we risk running out of ideas and methods for creating the 'unprecedented' software of the future that will maintain our global competitiveness and national security. To help in the construction of real-world no-surprise systems, theoretical research should be sensitive to the issues raised by empirical analyses and to the scalability problem.
- "Address human needs significantly better as we engineer the large, unprecedented systems of the next century subject to concurrent safety, evolvability, and resource constraints.
- "Form teams to build important advanced applications that will both serve as testbeds for the new ideas and address a serious problem identified by the PITAC: 'desperately needed software is not being developed.'"

After the workshop, questions arose about whether this report recommends a "software engineering" or a "software research" agenda, and how software research should address such areas as operating systems, networking, artificial intelligence, and database software.

USC WORKSHOP FINDINGS

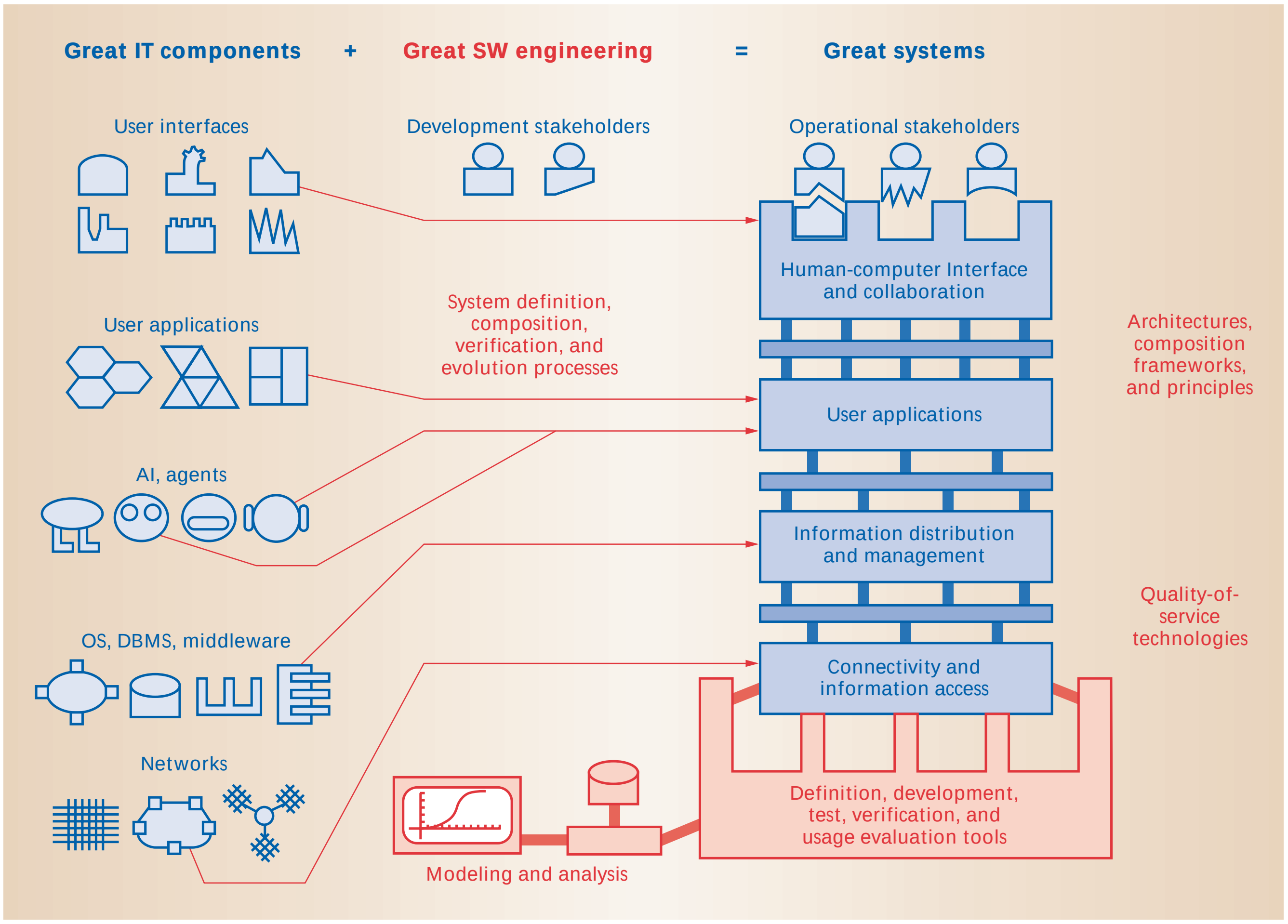
To clarify the nature and role of software engineering in IT research and to identify software engineering research strategies, a second meeting—the NSF Software Engineering Research Strategies Workshop—took place at the University of Southern California in August 1999. Participants presented their findings to the NSF a month later.

The USC meeting participants felt that understanding the proper place of software engineering research meant beginning with a discussion of the future. They saw an enormous increase in complexity over the horizon as every conceivable device becomes connected to everything else—not only computers and trains, planes, and automobiles, but also appliances of all types, from refrigerators to information generators and receivers. Also, with increasing dependence on independently evolving commercial components, control over content diminishes rapidly, raising important technical and legal issues.

The fast new global delivery mechanism also means that time-to-market considerations are becoming paramount. Product cycles that only recently spanned an already high-pressured 18 months have dropped to six months or less. Yet, as a consequence, fantastic opportunities have emerged at all levels.

Synergizing IT and software engineering

Participants noted that if an organization wants a good information-technology-based system, there must be a synergetic relationship between the IT components that comprise the delivered system and the software engineering elements that guide its definition, development, component selection, integration, and validation. You cannot go from good to great IT systems if you improve only the SE elements and neglect the IT components. As Figure 1 shows, it is equally true that focusing only on great IT components without attention to complementary SE practices generally leads to failure as well.[2]



Great IT components
+
Great SW engineering
=
Great systems
User interfaces
Development stakeholders
Operational stakeholders
Human-computer Interface and collaboration
System definition, composition, verification, and evolution processes
User applications
Architectures, composition frameworks, and principles
User applications
AI, agents
Information distribution and management
Quality-of-service technologies
OS, DBMS, middleware
Connectivity and information access
Networks
Definition, development, test, verification, and usage evaluation tools
Modeling and analysis

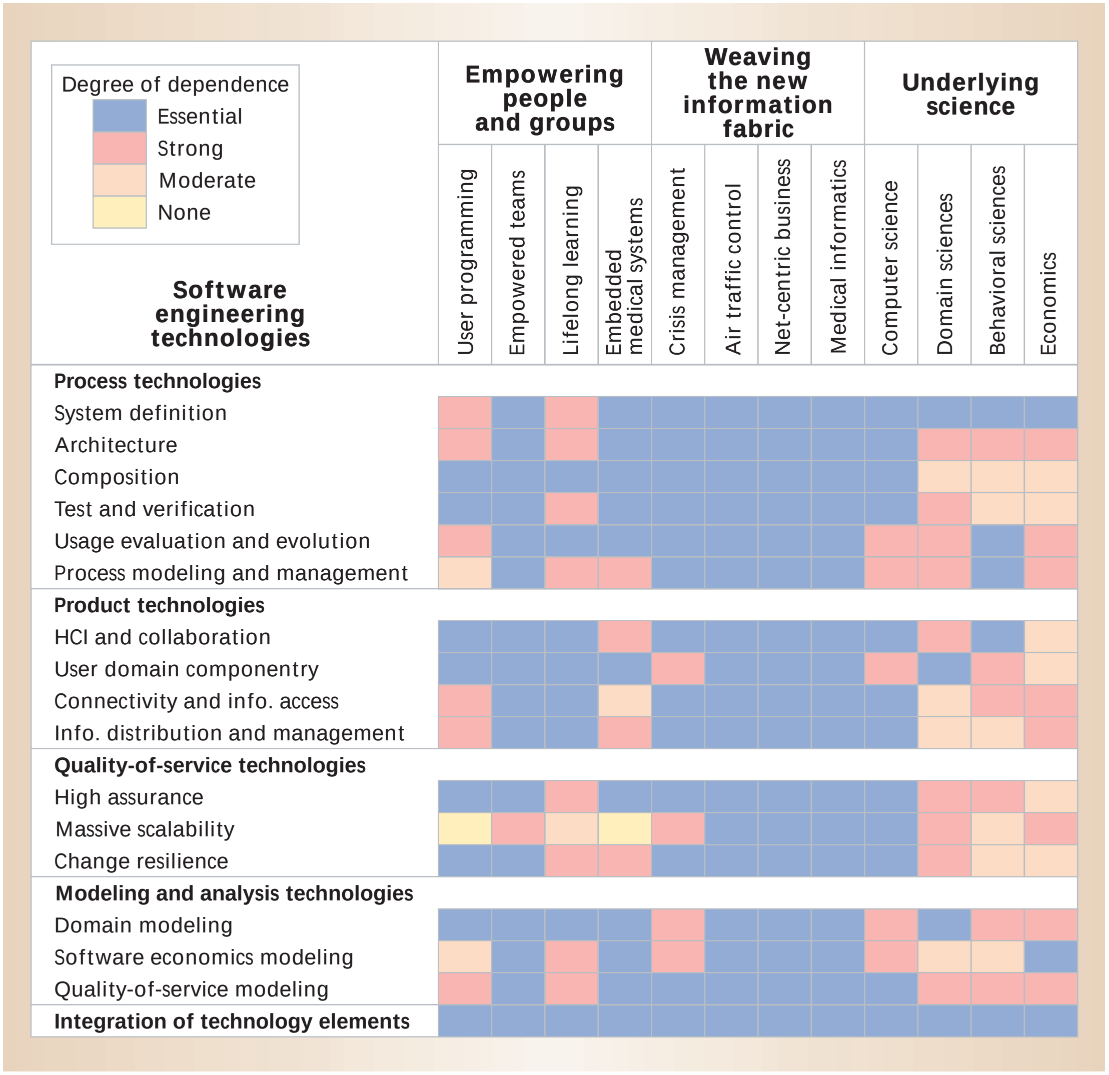
Figure 1. An organization that wants a good IT-based system needs a synergetic relationship between the IT components (blue), which comprise the delivered system, and the software engineering elements (red), which guide the system's definition, development, component selection, integration, and verification.

A good example of the latter approach is the AESOP project experience.[3] This project assumed that a small number of advanced IT components—a user-interface generator, an object-oriented DBMS, and two middleware components—could be integrated quickly and cheaply: in six months, by only two very bright software engineering researchers. It was a "simple matter of programming." The result: a schedule overrun by a factor of four, an effort overrun by a factor of five, and slow, unresponsive system performance. Similar approaches on large-scale systems generally experience even worse results,[4] but often remain undocumented.

Fortunately, the AESOP experience was well analyzed by the SE researchers, who identified architectural mismatch as the key success inhibitor. This analysis led to a rich experience-based SE research program that addresses and copes with the sources of architectural mismatch.[5]

Creating stronger and better-integrated IT components and SE capabilities to achieve great, no-surprise IT systems would be a tremendous challenge even if software and IT remained static. But given the constantly increasing pace of change, the challenge becomes overwhelming. The Web and the Internet connect everything with everything else. Autonomous agents that make deals in cyberspace create many opportunities for chaos, and systems of systems, networks of networks, and agents of agents create huge intellectual-control problems.

Further, the economics of software componentry leave system developers with no choice but to incorporate large commercial-off-the-shelf components into their systems. Unfortunately, developers have no way of knowing what is inside these COTS components, and they have no control over the direction of their evolution. Software architecture and COTS decisions are made hastily, and the time pressures forcing this "marriage in haste" leave no opportunity for repenting even at leisure.



Degree of dependence
- 🟦 Essential
- 🟥 Strong
- 🟧 Moderate
- 🟨 None

Software engineering technologies	Empowering people and groups				Weaving the new information fabric				Underlying science			
	User programming	Empowered teams	Lifelong learning	Embedded medical systems	Crisis management	Air traffic control	Net-centric business	Medical informatics	Computer science	Domain sciences	Behavioral sciences	Economics
Process technologies												
System definition	🟥	🟦	🟥	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
Architecture	🟥	🟦	🟥	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥
Composition	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟧	🟧	🟧
Test and verification	🟦	🟦	🟥	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟧	🟧
Usage evaluation and evolution	🟥	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟦	🟥
Process modeling and management	🟧	🟦	🟥	🟥	🟦	🟦	🟦	🟦	🟥	🟥	🟦	🟥
Product technologies												
HCI and collaboration	🟦	🟦	🟦	🟥	🟦	🟦	🟦	🟦	🟦	🟥	🟦	🟧
User domain componentry	🟦	🟦	🟦	🟦	🟥	🟦	🟦	🟦	🟥	🟦	🟥	🟧
Connectivity and info. access	🟥	🟦	🟦	🟧	🟦	🟦	🟦	🟦	🟦	🟧	🟥	🟥
Info. distribution and management	🟥	🟦	🟦	🟥	🟦	🟦	🟦	🟦	🟦	🟧	🟧	🟥
Quality-of-service technologies												
High assurance	🟦	🟦	🟥	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟧
Massive scalability	🟨	🟥	🟧	🟨	🟥	🟦	🟦	🟦	🟦	🟥	🟧	🟥
Change resilience	🟦	🟦	🟥	🟥	🟦	🟦	🟦	🟦	🟦	🟥	🟧	🟧
Modeling and analysis technologies												
Domain modeling	🟦	🟦	🟦	🟦	🟥	🟦	🟦	🟦	🟥	🟦	🟥	🟥
Software economics modeling	🟧	🟦	🟥	🟦	🟥	🟦	🟦	🟦	🟥	🟧	🟧	🟦
Quality-of-service modeling	🟥	🟦	🟥	🟦	🟦	🟦	🟦	🟦	🟦	🟥	🟥	🟥
Integration of technology elements	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦

Figure 2. Software engineering technologies, mission challenges, and underlying science. Each column in the chart addresses a functional area, with each row listing a specific SE technology. A given functional area's degree of dependence on a specific technology ranges from essential (blue) to none (yellow).

Having explored the nature and contributions of software engineering research, the workshop turned to the problem of characterizing software engineering elements and their individual roles in realizing various classes of future IT systems.

Focus on two challenge areas

Participants began their characterization of SE research elements by reviewing and discussing two excellent sources of overall software Grand Challenge problems.[6] With the aid of these sources, we defined two SE-intensive challenge areas for achieving the quality-of-life improvements envisioned in the PITAC report. These areas involved harnessing future IT in "empowering people and groups" and in "weaving a new information fabric" that is much more reliable, supple, and adaptable than current technology.

Within each of these two challenge areas, we identified four specific sample applications, and characterized their potential future IT- and SE-enabled capabilities and the relative importance of individual SE research areas in achieving those capabilities. We then aggregated these parameters into a cross-impact matrix that clarified the relationships and led to a number of conclusions about the nature of SE research strategies, as Figure 2 shows.[7]

As one example, consider user programming. To achieve safe and effective user programming in the future, we face an essential dependence on achieving significant progress in

- safe and effective composition of components and applications;
- test and verification that the users' programs have no serious faults or adverse side effects;
- supportive human-computer interface technology;
- user domain componentry for composing user applications;
- high-assurance technologies for ensuring that user programs do not compromise reliability, privacy, availability, safety, mission performance, or organizational viability;
- change resilience to ensure that user modifications can be done quickly and with high assurance; and
- ensuring that each technology integrates well with the others (for example, that user domain componentry evolves in ways consistent with the change resilience assumptions).

Other software engineering technologies are less essential to successful user programming, primarily because the programs and projects are relatively small. We do not need massive scalability at all, and process and economics modeling contribute only moderately. The strong dependencies for the other technologies primarily address their need to establish sound support frameworks and environments for user programming—system definition, architecture, connectivity and information access, and information distribution and management.

We also considered each software engineering technology's relative degree of dependence on four primary underlying science areas: computer science, domain sciences, behavioral sciences, and economics. Suppose that a set of stakeholders sought to negotiate the definition of the best air-traffic-control system that can be built within a given budget and schedule. The system definition support capabilities would depend in an essential way on a knowledge of computer science (the performance of algorithms); domain sciences (the computations needed to predict clear air turbulence); behavioral sciences (for both stakeholder requirements negotiations and air-traffic-controller group performance); and economics (for performing cost-benefit analyses of various alternatives).

Determining the most appropriate logical and physical IT architecture for the air-traffic-control system would involve a strong dependence on all four underlying sciences, but the dependence on getting the best information structures would be considerably more essential for computer sciences than for the other sciences.

Primary concerns

We can draw many conclusions about software engineering research strategies from the preceding analysis.

- *The field's needs are significant.* Each of the eight sample challenge applications exhibited essential or strong dependencies on improved capabilities in most software engineering technology areas. Having integrated, mutually reinforcing technology elements was essential for all the challenge applications. Having a good product or a good process technology alone makes producing a good system unlikely; you must have both.
- *The field's needs are interdisciplinary.* System definition technology, architecture technology, and the other SE technologies require more than traditional computer science to ensure successful IT applications. This truth does not imply that single-discipline research is unimportant. But it does mean that the body of knowledge required for successful software engineering includes considerably more than computer science.
- *There is no silver bullet for success.* Our discussions corroborated Fred Brooks's thesis[8] that no silver-bullet solution exists. We concluded that a balanced portfolio of research investments and an emphasis on integration of software engineering and information technology solutions via experimental application are most likely to show progress toward addressing the challenges.
- *Developers must "skate to where the puck is going."* Workshop participants felt that too much research focuses on past problems. Examples of future trends where we need more research into problems of intellectual control include heterogeneous distributed systems, dynamically changing software structures, and interactions among autonomous agents.
- *Developers need to explore new metaphors for software composition and evolution.* One approach to future problems looks for new perspectives or metaphors for intellectual control. Examples include biologically self-testing or self-adaptive software systems and the socioeconomic employment of software goal and reward structures.

Changing development processes or adopting a software product line requires that people change their behavior on projects, often for payoffs only seen much later.

Participants expressed serious concern about software engineering technology transition, which has always been exceptionally difficult. Adopting new practice or technology involves behavioral change and deferred gratification. You can plug in a faster chip or algorithm without changing your project practices, and you can see the effect on performance immediately. Changing development processes or adopting a

The primary difficulty in determining appropriate metrics is that software projects are effectively unrepeatable in practice.

software product line requires that people change their behavior on projects, often for payoffs only seen much later and that are hard to trace definitively. Collaborative university-industry research and experimentation can expedite transition. Several good suggested mechanisms for pursuing this approach can be found elsewhere.[1]

Relating software research results to key challenge applications will provide greater understanding about which processes, architectural styles, and IT components best fit such applications. This work provides the foundation for no-surprise projects in other application domains and the ability to reason about what level of capability can be achieved within a given time or budget.

Keeping track of progress toward this objective will help clarify which challenge problems lie outside the no-surprise boundary, providing stakeholders with more realistic expectations about encountering problems. Tracking progress will also enable stakeholders to consider alternative strategies for bounding risk, such as using delivery time or cost as their independent variable and desired features as the dependent variable.

Applying metrics to software development

Finally, workshop participants also discussed the formulation of appropriate metrics for software development progress and their use in evaluating the effects of advances in software engineering research. The participants concluded that this task's difficulty should not be a barrier to doing it better.

The primary difficulty in determining such metrics is that software projects are effectively unrepeatable in practice. We find differences among team skills and dynamics or among learning-curve effects on similar products developed by the same team. Increasingly, rapid changes in the IT marketplace and infrastructure continually change the rules of software development from one project to the next.

This changing nature means that most traditional measures of software productivity are increasingly irrelevant. A good example is the metric of new source lines of code produced per person-day on a large project. The value of this metric continues to hover around 10, giving the impression of no progress. However, analyzing several decades of experience at Bell Laboratories[9] demonstrates that the number of executing lines of machine code generated by a line of source code has increased by roughly an order of magnitude every 20 years.

Even stronger progress indicators can be generated by combining productivity per IT platform with the number of platforms in use. Effective examples in the computer hardware and communications field include plots showing high exponential growth in number of transistors in use, in petaflops of computing power available, or in numbers of Internet packets handled per year. A software counterpart to these examples, using similar counting rules, is lines of code in service (LOCS), obtained by multiplying executable lines of machine code per computing platform by the number of platforms in use. As observed elsewhere,[10] over several decades the US Department of Defense's LOCS have increased by an order of magnitude roughly every seven years, with cost per LOCS currently decreasing at about the same rate.

Metrics such as LOCS, transistors, petaflops, and packets at least pass a market test for value because they are items that people and organizations have paid market prices to obtain. Beyond this, however, we would prefer metrics that more closely reflect value added to people and organizations. Such metrics are emerging for software, but face challenges in applying them across different application areas and across stakeholders having different values and priorities.

The automated collection and analysis of usage-experience data from human and computer sources presents a significant challenge in "weaving the new information fabric." Soon, we will be able to sample and analyze billions of concurrent transactions. This capability can be harnessed not only for better personal, business, and government decisions, but also for analyzing and improving the various dimensions of effectiveness in the software and IT systems we produce in the future.

POTENTIAL ROADBLOCKS

Although the two workshops held in 1999 helped establish a firm foundation for the NSF's research efforts, challenges remain. For example, any company that hopes to succeed in today's fiercely competitive market cannot spend too much time building quality in. Doing so will cost it so much market share that its product will fail when it finally reaches the marketplace, even if the software is more robust than its competitors. Thus, a major NSF goal is to devise a technology that lets developers accelerate the software development process without compromising quality.

On the other hand, some analysts believe that the race-to-market model may evolve toward a slower, more quality-focused paradigm when a given market matures. Some US analysts, for example, speculate that CMM (Capability Maturity Model) Level 5 companies in India could someday build more robust versions of popular software products, thereby reprising the role Japanese automakers played in the 1970s when their less-expensive and better-engineered cars challenged domestic auto manufacturers' market dominance in the US. Opinions differ on how likely this scenario is, however. History didn't repeat itself, for example, when the Japanese software factories went head-to-head with less structured but more agile US companies. But quality-focused developers may yet discover an effective way to compete.

During the USC workshop, several participants discussed how we might leverage some of the techniques the open source community uses to build robustness into software. Aside from concerns about devising a workable economic model, participants noted that the open source model works best when it supports a product with broad enough appeal to attract a critical mass of contributing programmers. Thus, what works for Linux or Apache may not be practical with niche applications that have much smaller user bases.

Creating a smooth interface between government and free enterprise also raised some concerns at the workshop. Cultural differences, for example, may arise between fast-paced software developers who place a premium on speedy progress and government officials who must work within stringent regulations.

Abstract by nature, software's apparently limitless flexibility is both its greatest strength and greatest weakness. We can no longer afford to let this increasingly critical component of the Information Age's infrastructure proliferate in the form of ill-conceived, hastily crafted, and failure-prone products. The NSF has taken the first steps toward grasping the reins of software development and regaining control not only of the development process but also of the role software will play in our future. ✷

References

1. Final Report, "NSF Workshop on a Research Program for the 21st Century," http://www.cs.umd.edu/projects/SoftEng/tame/nsfw98/.
2. "Role of Software Engineering in IT Research and Systems," chart 5, http://sunset.usc.edu/Activities/aug24-25/NSFtalks/Boehm.ppt.
3. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse Is So Hard," *IEEE Software*, Nov. 1995, pp. 17-26.
4. Standish Group, "CHAOS," http://www.standishgroup.com/chaos.html.
5. M. Shaw and D. Garlan, *Software Architecture*, Prentice Hall, Upper Saddle River, N.J., 1996.
6. J. Gray, *What Next? A Dozen Information-Technology Research Goals*, (draft Turing lecture), tech. report MS-TR-99-50, Microsoft, Redmond, Wash., June 1999.
7. "Software Engineering Technologies, Mission Challenges, Underlying Science," chart 12, http://sunset.usc.edu/Activities/aug24-25/NSFtalks/Boehm.ppt.
8. F.P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, Apr. 1987, pp. 10-19.
9. L. Bernstein, "Software Investment Strategies," *Bell Labs Technical J.*, Summer 1997, pp. 233-242.
10. B. Boehm, "Managing Software Productivity and Reuse," *Computer*, Sept. 1999, pp. 111-113.

Barry Boehm is director of the USC Center for Software Engineering. He developed the Constructive Cost Model, the software process Spiral Model, and the Theory W (win-win) approach to software management and requirements determination. Contact him at boehm@sunset.usc.edu.

Victor R. Basili is a professor in the Institute for Advanced Computer Studies and the Computer Science Department at the University of Maryland. He has participated in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial and government settings and has consulted with many agencies and organizations. Contact him at basili@cs.umd.edu.