Scaling Up: A Research Agenda for Software Engineering

The following excerpts have been gleaned from a report by the Computer Science and Technology Board that summarizes the deliberations of a group of software engineers participating in a CSTB workshop that focused on setting research priorities.

Business, government, and technical endeavors ranging from financial transactions to space missions increasingly require complex software systems to function. The complexity of the software arises from stringent requirements (e.g., for reliability in performance and integrity of the data used), the need to support a range of interactions with the environment in real time, and/or certain structural features (see Table I). These attributes make software difficult to produce and often require large-scale projects involving the efforts of several hundred—even a few thousand—programmers, software engineers, applications experts, and others working for one year or more. The difficulty of developing complex software systems tends to force delays in the implementation of new applications, compromises in what those applications can do, and uncertainties about their reliability. As a result, there is a perception in the field, especially in industry, that opportunity costs are large and growing.

How can this situation be improved? The problem has resisted the efforts of many talented individuals over many years. Some degree of resistance to change is inevitable, reflecting the inertia that comes from the large and cumulative investment that companies have made in their software development processes. However, the Computer Science Technology Board (CSTB) workshop participants expressed a widely shared frustration that options circulating within the software engineering community fall short of what is needed (or fall on deaf ears). In the past suggested solutions have ranged from ways to improve tools used by software developers to ways to improve the management of software development teams; they have often been couched as options for improving productivity in software development, itself a slippery and many-sided concept.

AGENDA RESEARCH

Directions for Change

Acknowledging those suggestions and accepting that there may be no "silver bullet" in this area [13], CSTB workshop participants reached the consensus that progress will be made if the vast array of existing and emerging knowledge can be codified, unified, distributed, and extended more systematically. Software requirements for large and complex systems have been outrunning understanding of their fundamental principles, and software development practices have not filled that gap. Further, the shared framework of existing knowledge has not grown commensurately with advances made by individual practitioners. Codification of existing knowledge would help to make the process of developing routine software (which is what most software is) more routine, thereby saving time and money. It is essential for progress in the reuse of knowledge. A strategy for dissemination of codified knowledge should build on the concept of software engineering handbooks.

CSTB workshop participants agreed that software engineering research can contribute to the improvement of practice if the research community broadens its view of what constitutes good research (and amends its reward structure accordingly). In particular, researchers need to look to practice to find good research problems, validating results against the needs of practice as well as against more abstract standards. The problems experienced by practitioners are serious and pressing, and they call for innovative solutions. The promise of fruitful interactions between researchers and practitioners should not have to founder because of cultural differences between the two groups.

The CSTB workshop underscored the need for both software engineering researchers and practitioners to accept a more realistic view of the problem. Many of the problems experienced today reflect implicit as-

TABLE I. Observations on Complex Systems

What makes complex systems complex? Is complexity inherent when software is the focal point of the system? How can we characterize the complexity of software systems? Can we identify avoidable and unavoidable kinds of complexity? Looking at systems from a broad perspective, we see several ways in which they may be complex:

Structure: Subsystems, modules, macros, down to statements and expressions

Behavior: Observable activity of a system
- **Function:** Transformations on components of the state
- **Process:** Flow of control
- **Reactivity:** Events to which the system must respond
- **Timing:** Constraints on response times

State: Persistent and transient data with associated consistency and correctness invariants

Application: Requirements from the context in which system will be used
- **Recovery** of state and continuation of reactivity
- **Security** of state from destruction or unwanted inspection
- **Safety** from catastrophic external or internal events
- **Interfaces** with users and other systems
- **Operations** that maintain the state and continuity of the system

Development environment: People and processes producing the code
- **Design and implementation**
- **Documentation and training**
- **Verification, validation, and certification**

SOURCE: Adapted from "Complexity, Multiple Paradigms, and Analysis," position statement by Susan L. Gerhart.

sumptions that the flow from software system concept to implementation is smoother and more orderly than it is, as well as implicit assumptions that a development process involving project teams is subject to the degree and kind of control that might be found if a single individual were responsible for the software. A clearer understanding of the realities of software development can lead to improvements in any of several ways. For example, it may facilitate the identification of valuable tools that do not now exist, or it may facilitate the identification of fundamental flaws in the software development process itself. A more realistic view will also make clear the extent to which problems are technical or are amenable to technical solutions.

Specific Short- and Long-Term Actions

Improving the development of complex software systems requires a series of long-term (5 to 10 years or more) and short-term (1 to 5 years) measures. The CSTB workshop reached consensus on several directions for change in software engineering research, which fall into three interconnected areas: (1) perspective, (2) engineering practice, and (3) modes of research (see Table II). These improvements are outlined and discussed in greater detail in the body of the report. Carried out together, they will bring to software engineering strengths found in traditional engineering disciplines, notably the effective reuse of shared knowledge and a working distinction between routine and innovative design. Some of these directions have been advanced before; others are more novel. CSTB workshop participants shared the hope that this new presentation and the consensus it represents will help to catalyze much-needed changes.

The remainder of this report examines each research agenda item in turn. Determining the details of implementing the measures outlined here was beyond the scope of the CSTB workshop. It is clear, however, that implementation of the recommended measures will hinge on funding for corresponding research projects and other incentives.

PERSPECTIVE

The software research community has not kept up with the development of complex software systems in industry and government, while in the field, managers concerned with procuring software systems often evince idealized and unrealistic views of how software is developed. A critical reality in the field that is insufficiently appreciated by academic researchers and systems purchasers is the extent to which existing systems tie up resources. So-called system maintenance may constitute up to 75 percent of a system's cost over its lifetime. The high cost of maintenance diminishes the amount of money available to replace existing systems, and it lengthens the payback period for investment in new system development. It makes designing for a system's entire life cycle imperative, but such design is rarely if ever achieved. CSTB workshop participants agreed that to progress in system development, it is time to portray systems realistically. They also agreed that a more rigorous use of mathematical techniques can help researchers to manage and diminish complexity.

[1] These conclusions are consonant with those of the Defense Science Board Task Force, [18] which focused on management aspects because attitudes, policies, and practices were a major factor in defense software system acquisition.

SHORT-TERM ACTIONS

Portray Systems Realistically

View Systems as Systems, not as Collections of Parts. While the computer field has helped to popularize the word "systems" and the concept of systems, it is ironic that information systems developers have not developed formal mechanisms to understand systems and the interrelationships among system components. Software engineering researchers have been unable to provide effective guidance to practitioners regarding the process of system definition and the concomitant implementation of functional elements. Progress in developing software systems requires a fundamental appreciation that those systems are more than just a collection of parts and that software is embedded in larger systems with a variety of physical components; design of such systems must deal with both of these issues. Design of software systems must also take into account the fact that the whole system includes people as well as hardware, software, and a wide variety of material elements.

Recognize Change as Intrinsic in Large Systems. Software projects are increasingly likely to be built on top of an existing, installed base of code rather than built anew. As that installed base of software grows over time, software systems that might or might not have been designed to endure have been patched, modified, and "maintained," transforming them greatly from their original designs and functions [5]. Two factors are at work here: The first is that systems are often not designed well-enough to begin with, and the second is that user needs change over time—new requirements arise, and existing systems must be adapted to accommodate them. But commonly used conceptualizations, such as the waterfall model or even the spiral model, assume a more sure-footed progression from requirement specification to design to coding to testing and to delivery of software than is realistic [10], [30]. Given that 40 to 60 percent or more of the effort in the development of complex software systems goes into maintaining—i.e., changing—such systems [9], the design and development processes could be made more efficient if the reality of change were accepted explicitly.

Sustaining the usefulness of software systems differs from the care of other assets because it entails two distinct activities: (1) corrective and preventive maintenance, which includes the repair of latent defects and technological wear and tear, and (2) enhancement, which normally introduces major transformations not only in the form but also in the functions and objectives of the software. Enhancement activities have been observed to constitute perhaps 75 percent of the total maintenance effort.[2]

The degree and impact of change is analogous to the evolution of an urban neighborhood: Over time, old and obsolete buildings are torn down, the supply of utilities changes in both quality and delivery aspects, and transportation routes and media change. As new needs, wants, and capabilities emerge, the structure and function of the neighborhood evolve. The neighborhood is not thrown out wholesale and replaced because doing so would be far too costly. As with changes in neighborhoods, changes in software are not always improvements; software systems suffer from the tension between providing for functional flexibility and assuring structural integrity of the system.

Software developers in industry and government are increasingly aware that change occurs from the earliest design stages as initial expressions of customer requirements are refined. Managing this change involves managing a mix of old code (typically with inadequate documentation of original specifications as well as modifications made over time), new programmers, and new technology. The process is *ad hoc*, and the problem grows over time; the larger the installed base of code, the more formidable the problem. The problem is aggravated where management decisions, including contracting decisions, keep developers and maintainers separate.

A critical reality in the field that is insufficiently appreciated by academic researchers and systems purchasers is the extent to which existing systems tie up resources.

Ideally, system designers leave hooks for the changes they can anticipate, but problems arise from major changes that result from changed circumstances or goals. Also, schedule and process pressures often militate against providing for functional flexibility and future changes. Further, the current generation of computer-aided tools for software engineers concentrates on development activities and generally neglects maintenance. As a result, supporting information for new code and the tools to exploit it are not carried forward from development to maintenance. More seriously, these tools do not accommodate large bodies of code developed without using the tools, although some progress is being made in the necessary restructuring of programs to accommodate computer-aided tools.

Just as change should be accepted as a basic factor

[2] The National Bureau of Standards (now the National Institute of Standards and Technology) drew on several studies to decompose maintenance into corrective maintenance (20 percent), including diagnosis and fixing design, logic, or coding errors; adaptive maintenance (25 percent), which provides for responses to changes in the external environment; perfective maintenance (50 percent or more), which incorporates enhancements; and preventive maintenance (5 percent), which improves future maintainability and reliability [22], [34]. Similarly, experience with U.S. Air Force weapons systems suggests that while 15 to 35 percent of software maintenance corrects design errors, 25 to 50 percent adds new capability, 20 to 40 percent responds to changes in the threat, 10 to 25 percent provides new system interfaces, 10 to 20 percent improves efficiency, 5 to 15 percent improves human factors, and 5 to 10 percent deletes unneeded capability [26].

in most large, complex systems, designing for change should become a fundamental body of knowledge and skill. The very notion of maintenance as an activity separate from the creation process seems to legitimize high costs, poor support, and poorly managed redesign. Eliminating this notion via a move toward designing and building systems in anticipation of change would help to increase the engineering control over post-release modification. Since software reflects both system specifications and design decisions, changing either element will indirectly produce changes in the code. One possibility is to strive for designing systems that are more modular or easier to replace as needs change.

It should be noted that the issue of determining what the software shall do (the "requirements definition") is much broader than software engineering practices today would suggest. This perceptual difference contributes to the maintenance problem. What is needed is a thorough investigation, analysis, and synthesis of what the combined functions will, or should, be of the automated and non-automated (human, business, or physical) elements of the system, including all "think flows," work flows, information flows, and other functionalities. A total systems approach would be involved with a heavy emphasis on the conceptualization of the functional role of both the automated parts and the fully combined systems, allowing for reengineering to accommodate or exploit the changes that are made possible by introduction of the automated system.

Understanding the reasons for change and the costs, impacts, and methods of change could lead to more control of a major part of software development costs. Creating mechanisms that allow for change and that make systems robust while undergoing change will help to reduce opportunity costs in system development and deployment. Part of what is needed is a change in attitude. But for the long term, a theory of software systems is needed that will build on empirical study of software system applications.

Study and Preserve Software Artifacts

Although systems developers work with an evolving set of goals and technologies, they can still learn valuable lessons from existing systems—lessons about what led to success or failure and what triggered incremental or major advances. The history of computing is replete with instances in which identifying the intellectual origins of key developments is difficult or impossible because most advances, in their time, were not thought of as intellectual issues but instead were treated as particular solutions to the problems of the day. Most software specialists would be hard put to name "seven wonders of the software systems world" or to state why those wonders are noteworthy.[3] Meanwhile, the artifacts of such systems are disappearing every day as older equipment and systems are replaced with newer ones, as projects end, and as new applications emerge. Because almost all large software systems have been built in corporate or government settings where obsolete systems are eventually replaced, and because those systems have received little academic attention, useful information may be vanishing.

A concerted effort is needed to study (and in some cases preserve) systems and to develop a process for the systematic examination of contemporary and new systems as they arise. Immediate archival of major software artifacts, together with the software tools needed to examine them, or even to experiment with them, would enable both contemporary and future study. Systematic study of those systems would facilitate understanding of the ontology of architecture and system components, provide a basis for measuring what goes on in software development, and support the construction of better program generators.

Studies of contemporary systems would provide an understanding of the characteristics of software developed under present techniques. Such an effort would examine software entities such as requirements documentation, design representation, and testing and support tools, in addition to the actual source code itself, which has traditionally been the focus of measurement. Better mechanisms that provide quantifiable measures of requirements, design, and testing aspects must be developed in order to understand the quality baseline that exists today. Existing mechanisms for measuring source code must be put to more widespread use to better assess their utility and to refine them [12], [23]. In addition, variations in quality need to be traced to their sources to understand how to control and improve the process. Thus this effort should encompass less successful as well as exemplary artifacts, if only to show how poor design affects maintainability. The examination of artifacts should be combined with directed interviews of the practitioners (and system users) and observation of the process to correlate development practices with resulting product quality.

Having quantifiable measurements would enable new, innovative development methods and practices to be evaluated for their impact on product quality. However, as the software industry evolves, so too must the measurement techniques. For example, if new means of representing requirements and design are put into practice, the measurement techniques must be updated to accommodate these new representations. In addition, efforts to automate measurement can be improved if researchers consider measurability as an objective when developing new development methods and design representations.

Such measurement and research cannot take place in the laboratory due to the size of the actual systems being developed (the costs of experiments at this scale are prohibitive). Moreover, it is unlikely that small experiments can be extrapolated to apply to large-scale projects. A cooperative effort between government, industry, and academia could provide necessary funding,

[3] An informal query addressed to a community of several hundred software engineering specialists suggested the following candidates: the SAGE missile defense system, the Sabre interactive system for airline reservations, the Yacc compiler tool for UNIX, ARPANET communications software, and the VisiCalc spreadsheet package, among others.

TABLE II. Agenda for Software Engineering Research

	Recommended Actions	
	Short Term (1–5 years)	**Long Term (5–10 years)**
Perspective	Portray systems realistically • View systems as systems • Recognize change as intrinsic Study and preserve software artifacts	Research a unifying model for software development—for matching programming languages to applications domains and design phases Strengthen mathematical and scientific foundations
Engineering practice	Codify software engineering knowledge for dissemination and reuse Develop software engineering handbooks	Automate handbook knowledge, access, and reuse—and make development of routine software more routine Nurture collaboration among system developers and between developers and users
Research modes	Foster practitioner and researcher interactions	Legitimize academic exploration of large software systems in situ Glean insights from behavioral and managerial sciences Develop additional research directions and paradigms—encourage recognition of review studies, contributions to handbooks

access to real-world artifacts and practitioners, and the academic research talent required for such an effort. Such a combined effort also would provide an excellent platform for greater collaboration in software engineering research between members of these communities. Designation and funding of one or more responsible entities are needed, and candidates include federal agencies (e.g., the National Science Foundation and the National Institute of Standards and Technology), federally funded research and development centers, or private institutions.

Finally, active discussion of artifacts should be encouraged. A vehicle like the on-line RISKS forum sponsored by the ACM, which provides a periodic digest and exchange of views among researchers and practitioners on risks associated with computer-based technology, should be established. Also, completed case studies would provide excellent teaching materials.

LONG-TERM ACTIONS

Build a Unifying Model for Software System Development

Shortcomings in software systems often reflect an imperfect fit to the needs of particular users, and in this situation lie the seeds for useful research. The imperfect fit results from the nature of the design and development process: Developers of complex software systems seek to translate the needs of end-users, conveyed in everyday language, into instructions for computer systems. They accomplish this translation by designing systems that can be described at different conceptual levels ranging from language comprehensible to the intended user (e.g., "plain English" or formal models of the application domain) to machine language, which actually drives the computer. Different spheres of activity, referred to by the profession as end-user domains, include the following:

- scientific computation;
- engineering design;
- modeling and visualization;
- transaction processing; and
- embedded command and control systems.

These domains tend to have different types of abstraction and different language requirements arising from differences in the representations of application information and associated computations. As a result, software developers work with a variety of domain-specific models. During the design process in any domain, key pieces of information or insights tend to be lost or misinterpreted.

How can the process of moving from a domain-specific model to a working piece of software be improved? One approach would be to develop a unifying view of the software design process and the process of abstraction, a view that would define a framework for the task of the complex software system developer. CSTB workshop participants did not reach a consensus on this complicated issue, but to illustrate the point, they began to sketch out the parameters for such a framework (Figure 1). For example, a system design can be thought of as a sequence of models, one ($M_{i,j}$) at each level. Different sorts of details and design decisions are dealt with at each level. The model at each level is expressed in a language ($L_{i,j}$). Languages are not necessarily textual or symbolic; they may use graphics or even gestures. Also, languages are not always formally defined.[4] Just as the domains of discourse at each level are different, so are the languages. Finally, the unifying view

[4] To serve this model-definition role, a language must provide five essential capabilities: (1) component suitability—module-level elements, not necessarily compilation units, with function shared by many applications; (2) operators for combining design elements; (3) abstraction—ability to give names to elements for further use; (4) closure—named element can be used like primitives; and (5) specification—more properties than computational functionality, with specifications of composites derivable from specifications of elements.

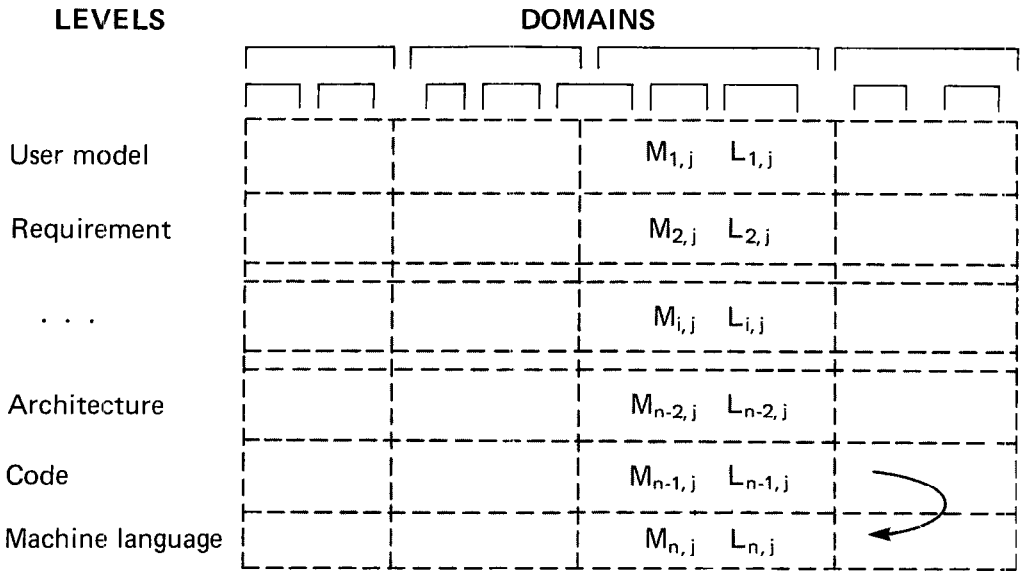

FIGURE 1. Illustration of a Unifying Model for Software System Design

would distinguish domain-specific models, a multilevel set of appropriate languages (although it is possible that languages may be shared—or largely shared—across domains), abstractions, and interlevel conversion mechanisms.

Useful Outcomes. The concept of a unifying model and the associated issues are emblematic of the larger problem of achieving more complementarity between software engineering research and practice. A unifying model would not necessarily be of immediate use to a system builder. But it would be a tool for academic analysis that could, in turn, yield structures and tools useful to a practitioner. In particular, it could help a researcher to analyze the software development process and forge improvements that might make that process more efficient in practice.

For example, a unifying view could help the software engineering researcher see the relation among existing mechanisms, or what mechanisms are missing, or devise ways to facilitate transitions from one major conceptual level to another (since it is necessary to be able to convert a system description at one level to a system description at an adjacent level).[5] By showing how parts are related, unification may facilitate the collapse of domain-specific models to include fewer levels than at present.[6] Eventually, it may be possible to move automatically from a description of requirements to a working product, bypassing intermediate levels. The modeling process can facilitate this progress much as the modeling of production processes in manufacturing has facilitated the reduction of the number of tasks in manufacturing processes and the application of manufacturing automation. As noted, better coordination technology, which would support both the modeling and the development processes would also be useful.

Research Implications. While the theory and nature of program transformation functions, drawing on a body of knowledge about language that crosses levels (sometimes called wide-spectrum language), have already been developed ([3], [28], [33]), the proposed kind of unifying view would also motivate new styles of research independent from those noted previously. Relevant current research addresses traditional programming language (although some of this research is in eclipse), computer-supported cooperative work (beyond the mere mechanical aspects—see Nurture Collaboration), and efforts to raise the level at which automation can be applied. Also needed are the following:

- Research that would support the development of domain-specific models and corresponding program generators—it is critical to recognize the legitimacy of specialization to the domain at the expense of expressive generality.
- Research to identify domains, levels, and commonalities across domains, since languages are needed for each level and domain.
- Research into the architectural level, which cuts across individual domain models. This level deals with the gross function of modules and the ways they are put together (for procedure call, data flow, messages, data sharing, and code mingling). The aggregates defined at this level include "state machine,"

[5] This process of transition is sometimes accomplished manually and sometimes mechanically. Mechanical transitions (e.g., using program generators or compilers) can enhance productivity, but they depend on more precision and understanding than are often available.

[6] Overall, the number of levels has grown and shrunk with technology over time. For example, today few people actually code in machine language, and relatively few program in assembly code.

"object-oriented system," and "pipe/filter system." Contrast this with the programming level, where the issues are algorithms and data structures and the defined entities are procedures and types.

- Research into whether it is possible to implement a concept found in the mechanical engineering environment, the quarter-scale model, and if so, how. A quarter-scale model, which would provide a more precise and detailed correspondence to the desired system than does a conventional prototype, would help to convey the complexity and various design attributes of a software system. It would allow practitioners to better comprehend how well a design works. It would also allow managers to control risk by helping them to understand where problems exist and to better estimate the resources required to solve those problems. In essence, it would make a seemingly intangible product, software, more real.

- Investigation of the mechanisms for making the transition between and among the various levels of abstraction. This research would involve exploration of automation aspects (e.g., compilers and generators) and computer-aided and manually directed steps. It would also involve exploration of the order of development of the models of a system: Whereas the conventional waterfall life cycle calls for completing each model before translating to the next, other approaches such as rapid prototyping or the spiral model allow for simultaneous development of several models.

- Reformulation of expressions of rigor and technical precision (sometimes referred to as "correctness"), performance given resources, traceability, cost, reliability, and integrity.

Strengthen the Foundations

In the absence of a stronger scientific and engineering foundation, complex software systems are often produced by brute force, with managers assigning more and more people to the development effort and taking more and more time. As software engineers begin to envision systems that require many thousands of person-years, current pragmatic or heuristic approaches begin to appear less adequate to meet application needs. In this environment, software engineering leaders are beginning to call for more systematic approaches: More mathematics, science, and engineering are needed [25].

Workshop participants focused on application of such approaches to software analysis. They also affirmed the value of mathematical foundations for better modeling and translation of real-world problems to the abstractions of software systems. Software analysis, which seeks to assure that software works as specified and as designed, is both a significant and a critical part of the implementation of large software systems. Unfortunately analysis activities have received too little focused attention, and what attention they have received has been largely limited to today's main analytical approach—testing. Testing techniques, moreover, are constantly being discovered and rediscovered.

A more rigorous and comprehensive approach to analysis is needed, one that renders techniques explicit, teaches about them, and develops its own literature and authority. In addition to testing, such techniques as proving, modeling, and simulation should be further developed and targeted to more properties (e.g., safety and functional correctness). Work is needed in performing measurements, establishing metrics, and finding a way to validate them. The understanding of what constitutes a defect and how to verify that designs or code are defect-free is today very limited.

In the absence of a stronger scientific and engineering foundation, complex software systems are often produced by brute force . . .

It should be noted that the ability to find defects earlier in the life cycle of a product or to prevent them from being introduced reduces test cost and reduces the number of defects in products delivered to end-users. This ability involves quality assessment and quality assurance. Research questions center on how to specify and measure the attributes (functional, behavioral, and performance) a system must possess in a manner that permits correct generation or proof. What aspects of a product can be assured satisfactorily only by testing as opposed to experimentation? What are the economic trade-offs between developing mathematical proofs and conducting testing? How to design for testability and verifiability is also an issue here.

Promising directions include the application of formal methods (which involve mathematical proofs), exploration of the mechanical and civil engineering concept of a quarter-scale model for previewing a design, application of the "cleanroom concept" (featuring walkthroughs of software with proofs of claims about features rather than checklists of flaws; [25]), and statistical quality control analogous to that used in manufacturing. A handbook of testing and/or quality assessment is desirable and will be possible with further development of the field of analysis.

ENGINEERING PRACTICE

The phrase "software engineering" was coined in 1968 as an expression of aspiration.[7] It remains today more an aspiration than a description.[8] The field of software engineering lacks the strengths and structure of other engineering disciplines, which have a more highly developed theory and firmer methodological foundations, as well as widely shared tools and techniques. Engineering disciplines are rooted in craftsmanship and evolve through a commercial stage (with emphasis on production and management) before becoming engineering as we generally know it (see Table III). What is needed is a way to define and discuss the "parts" of software engineering, the specifications for each, and a conceptual framework within which to place them. Organizing known techniques and information to identify and describe the parts of the software enterprise and how they fit together would go a long way toward enabling cleaner, more flexible design and development processes [8].

SHORT-TERM ACTIONS

Codify Knowledge for Dissemination and Reuse

Codifying existing software engineering knowledge and disseminating it through handbooks would help achieve several desirable ends: increasing the amount of software that can be created routinely, contributing to knowledge reuse, and ultimately, it is hoped, helping to reduce the size of programs, the time required to develop them, the risk of unacceptable errors, and the tendency to reinvent solutions to the same problems.

For software engineering to progress as a discipline, far more routine software development must be produced routinely. At a time when our needs for software are beginning to outstrip our ability to produce it, efforts to reduce the number of tasks requiring human effort are one obvious way to improve the situation. Practice in traditional engineering disciplines includes opportunities for both innovative design (creating things that have not been done before) and routine design (creating yet another example of a class of things that is well understood). Current software practice tends to treat most designs as innovative, even when knowledge exists that should render them routine. There is a need to make the reuse of knowledge routine, something many observers lament is far from happening.

> Indeed, if builders built buildings the way many programmers wrote programs, then most of us would still be homeless, because builders, like too many programmers, would be busy reinventing their technology every time they built something new. Continually having to rediscover carpentry, metallurgy, and project management, as well as having to write new building codes, would clearly be enormous disincentives to productivity. . . . [11]

Codifying knowledge and making it more accessible could be an important step in moving toward a situation in which machines can do some of the routine tasks, leaving those more complex and creative tasks to humans.[9] This is one potent way to improve software development productivity. Toward this end, academic researchers can help practitioners by developing a conceptual framework for software elements, routine designs, and standard components, much as chemical engineers have developed a framework for the reuse of design elements at a large scale [29].

Reuse of code, a less flexible concept than is reuse of knowledge, is the avenue for minimizing programming effort that has been most widely discussed in the software research and development community [8]. Although theoretically attractive, there are many barriers—both technical and sociological—to significantly improving the amount of reuse actually achieved. Achieving reuse involves more than building libraries of programs, and it requires research on what kinds of reuse are feasible, how to index, how to represent reusable elements, and how to deal with variations in the language in which a piece of reusable code is stated or even in the wording of the specification. But so-called code libraries serve as precursors to the broader concept of handbooks; current work in that area provides a useful starting point.

Develop Handbooks

Software engineering should follow the lead of other engineering fields, which codify basic knowledge and use handbooks as carriers of common knowledge, thereby reducing the tendency for dispersed practitioners to independently develop solutions to common problems, duplicating effort while diluting progress. Handbooks for such disciplines as mechanical and chemical engineering allow a broad sharing of general and specific technical knowledge, which provides a base for further progress. Software engineering needs such products; references during the CSTB workshop to heavily used copies of Don Knuth's multivolume work, *The Art of Computer Programming* [21], illustrate that a demand exists but remains unmet except in selected, narrow instances.

The structure and contents for software engineering handbooks cannot be determined without progress in accomplishing the codification discussed earlier. What is clear, however, is that there is a need for substantive as well as process knowledge to be conveyed in these handbooks, and it is that substantive component that

[7] The classic reference is to the software engineering workshop sponsored by the NATO Science Committee in Garmisch, West Germany, Oct. 7–11, 1968.

[8] In gross terms, software engineering is concerned with the practical aspects of developing software, such as design under various constraints and economic delivery of software products. It overlaps all of the various specialties of software research, including programming languages, operating systems, algorithms, data structures, data bases, and file systems, and it also addresses such cross-cutting qualities as reliability, security, and efficiency.

[9] Thus it could provide a foundation for exploration of user-programmable application generators, which may be appropriate for smaller systems.

TABLE III. Engineering Evolution

	Craftsmanship	Commercial Practice	Professional Engineering
Practitioners	Virtuosos and amateurs	Skilled craftsmen	Educated professionals
Practice	Intuition and brute force	Established procedure	Analysis and theory
Progress	Haphazard and repetitive	Pragmatic refinement	Scientific
Transmission	Casual and unreliable	Training in mechanics	Education of professionals

SOURCE: "Maybe Your Next Programming Language Shouldn't Be a Programming Language," position statement by Mary Shaw.

distinguishes these handbooks from the manuals that individual organizations use to standardize the software development procedures followed by their employees. Thus handbooks should contain a compendium of algorithms (for example, see [15]), test methods, and items pertaining to design and programming style. Also, to help practitioners work within the practical constraints they face, handbooks must vary for different domains; the languages, knowledge, and processes associated with, say, transaction processing systems differ from those used for large-scale scientific processing or other types of systems.

Given the dynamic nature of the field, a software engineering handbook should be one that can use computer technology to deliver its contents—an electronic handbook. The goal is to have a repository of information that creates a uniform organization for current knowledge, presents the information accessibly, and provides a means for updating its contents easily.

LONG-TERM ACTIONS

Automate Handbook Knowledge

To maximize the effectiveness of an electronic handbook, advances in several areas to make such products easy and attractive to use will be necessary. A research initiative aimed at the development of an electronically accessible, interactive software handbook should be inaugurated to develop the following:

- concepts and notations for describing designs and components;
- techniques for organizing and cataloging designs and components;
- techniques and representations for storing, searching, and retrieving designs and components;
- codification of routine designs and components for a large variety of types of software and applications;
- techniques for evaluating designs and components in terms of engineering trade-offs;
- techniques for modeling and simulating systems based on routine designs and components;
- criteria for evaluating and accepting or rejecting handbook entries; and
- technology to make the handbook easily usable and easily accessible.

If the technology and the electronic handbooks can be developed, it will be important to educate software engineers about appropriate methodologies and techniques for using the information they contain. The handbooks themselves will facilitate the teaching of routine design as part of software engineering—itself an important step toward increased productivity. Finally, the handbooks should not only be electronically "recorded," but they should also be built into the standard tools of software engineers, making for a truly activist incarnation.

Nurture Collaboration

Complex software systems are created by the efforts of many people—sometimes as many as a few thousand organized into multiple teams—and frequently no one person has a thorough understanding of the interaction of the entire system. Further, the software developers must communicate with end-users and others to understand the application, the issues, and the requirements. System development is an exercise in collaboration, and it is necessary to maximize the effectiveness of that collaboration. Although the team management problem has captured much attention and concern—much current software engineering consists of *ad hoc* measures or what could be called "crowd control"—today's measures do not go far enough [2], [6], [24].

Methodologies for iterative design are necessary. Specifications will always be idealized and simplified, and neither users nor designers are able to envision the full functionality of the resulting system during the traditional design stages. Consequently, system requirements are not so much analytically specified (contrary to appearances) as they are collaboratively evolved through an iterative process of consultation between end-users and software developers. Too many projects or designs have been completed that do not accomplish the desired end because substantive information was not well conveyed or understood in the design or implementation process [17].

Better linkage of knowledge about application areas or domains with software engineering expertise is essential; it is an important direction for exploration. Another involves developing and sustaining a common world view of systems under development. And a third is gaining understanding about how skilled designers make architectural trade-offs in the designs of systems [20], [32].

Better tools to support and enhance cooperative work are necessary in order to provide productivity enhancements; the more time that programmers can spend designing and programming pieces of systems that uniquely require their attention, as opposed to invest-

ing their time to overcome communications difficulties, the more likely it is that systems can be built in less time. Various forms of groupware, or tools for computer-supported cooperative work, may prove well suited to the collaborative process of system development. Also, the development of high-speed, ubiquitous computer networks, coupled with sophisticated and easy-to-use resources available through network access, may provide software engineers with valuable research and development tools [6]. For example, the growth of the information services business has illustrated the market potential of data base searching, and handbook implementation will depend critically on network access to data base facilities.

The call for improved collaboration is not new, nor are discussions about computer support for collaboration. But it may be particularly timely, since the motivation in this area is high and new tools are appearing and becoming more economical to use.

RESEARCH MODES

To complement existing directions in software engineering research and to better address the problem of developing software for large systems, CSTB workshop participants identified a need for cross-fertilization between academic software engineering researchers and practitioners as well as between software engineers and specialists in the behavioral and managerial sciences. CSTB workshop participants also urged universities to encourage additional topics and styles of software engineering research and to seek commensurate funding.

Much of the expertise in complex software systems resides in corporations, government research centers, and other nonacademic institutions.

SHORT-TERM ACTION

Foster Practitioner and Researcher Interactions

There is little academic investigation of the practices, techniques, or problems out in the field today. To rectify this situation, greater interaction among researchers and practitioners is needed as a first step. Such interaction has proved a boon in, for example, manufacturing engineering. Industry and university collaboration in that field has provided researchers and students access to real-world problems and constraints, while providing practitioners with access to creative problem-solving talent and new techniques.

The interaction of academia, industry, and government in software engineering has been inhibited by culture and tradition [7]. Although much is known about how complex software systems are built, there are few connections among the various repositories of practical knowledge. Much of the expertise in complex software systems resides in corporations, government research centers, and other nonacademic institutions. It is largely inaccessible to the academic community because of considerations of product delivery, proprietary knowledge, and cultural differences between the corporate and academic communities involved in software research.

That academic computer scientists do not often study large software systems and the process of developing them is one reason that practitioners often feel that the issues studied by academia do not adequately address the problems and challenges faced by builders of large systems—despite an apparently large body of systems analysis, systems design, and other university courses that do address systems issues. This is particularly so, for example, for complex systems involving software embedded in other products or systems (ranging from spacecraft to medical technology) and those systems that involve distributed processes in multiple nonhomogeneous computing and storage elements.

There are a number of reasons that information generated in our universities flows slowly only into the commercial sector: Academics do not study large systems because they do not have them or have access to them, and commercial and academic software specialists tend to read and have their work published in different journals. On the other hand, many topflight corporate researchers and developers, to the extent that they publish at all, do not publish in archival computer science journals because their topics—problems of practice—are not deemed scholarly.

The disparity in perspective and exposure existing between the academic software engineering research community and the practitioners of the corporate world hinders U.S. progress in developing complex software systems. Reducing that disparity is imperative, and it will require a greater degree of interaction between the two groups. Special meetings like the CSTB workshop are but a beginning to this process; implementing an initiative to preserve and study major artifacts and legitimizing academic exploration of large software systems are other vehicles for interaction.

LONG-TERM ACTIONS

Legitimize Academic Exploration of Large Software Systems

Academic investigation of research topics based on problems encountered in the real world by software developers could help industrial and other practitioners in both the short and long terms. For this to happen, new attitudes and incentives must be adopted.

As currently structured, most academic departments

are not conducive to large-system research. The tendency of universities to encourage and reward narrow specializations compounds the problem of a lack of opportunity or funding for access to large, complex systems by academic software researchers. Another side of this problem is the focus of the academic world on individual actions, whereas the corporate world is more team oriented. The realities of academic life—funding, tenure tracks, and other career concerns—militate against an individual academic researcher making a strong commitment to large-system research without consideration from the surrounding environment.

Further, whereas industry tends to focus on a problem as it appears in production, researchers (whether corporate or academic) need to find the underlying conceptual problems that are amenable to the development of knowledge that transcends a particular system manifesting a problem. Identification of good research problems based on production problems is a nontrivial problem that itself requires focused efforts. And to pursue that research requires analytical advances inasmuch as abstract formal models are lacking, language design issues are in eclipse, and testing and measurement have not been formalized.

ing managers and practitioners seek will come at a price: Industry must be willing to provide support—financial and human resources, and computer resources for experimentation—as well as access to the records of the proprietary system. Mechanisms would be needed to compensate industry for its efforts to produce data in a form useful to researchers or for bearing the risk of experimenting with novel development activities.

Perhaps the biggest concern is protecting the proprietary interests of corporations, for whom large systems are often a source of competitive advantage. Although the academic culture is devoted to openness and information exchange, universities are actively grappling with the problems of protecting corporate proprietary information that are presented by increasing corporate interest in research on practical problems. Business schools appear to have solved this problem some time ago. It should be possible to extend such efforts to apply to academic research into corporate software systems.

Finally, one way to get around some of the difficulties of studying large systems in corporate settings would be to facilitate the study of large systems in government settings. The federal government has been

One way to get around some of the difficulties of studying large systems in corporate settings would be to facilitate the study of large systems in government settings.

Funding is a major consideration. Funding of some considerable magnitude is needed if large systems are to be built—which is necessary to determine feasibility—and studied in academic settings, because the artifacts being studied are large. Also, while some universities have state-of-the-art hardware resources (although many do not), universities seldom invest in software tools and tend to lag behind industry in that area. This is a problem because there must be a fit between hardware and software across academic and industrial environments if large artifacts are to be experimented with other than as text (code). Thus it is difficult to study large systems cost effectively. Solving this problem requires innovations in funding, the details of which were beyond the scope of the workshop but which would clearly involve actions by government research funders, universities, and companies (including product development as well as research entities). Another direction for improvement and relief may come from enhanced networking, such as through the proposed national research network, which would allow dispersed researchers to share access to artifacts, other researchers, and practitioners [16].

If software systems are to be studied in corporate settings, a number of other difficulties will need to be overcome on the industry side. Resolving these difficulties will take much thought and concerted action; the CSTB workshop identified key directions for change. The insights and enhancements that software engineer-

the impetus for the development of large-scale integrated systems, interaction with academic researchers is a long-time tradition for many government organizations, and government entities are more obligated to respond to government programs or mandates. However, inasmuch as federal systems are developed and/or managed by private organizations, limitations on access to design and development processes and personnel may have to be overcome, as in purely corporate settings. Also, some peculiarities of federal systems development are not generalizable to commercial systems. For example, the federal procurement process is associated with specifications that are much more detailed than those typically generated by commercial buyers. Study of federal systems may therefore be an option that is second best.

Glean Insights from Behavioral and Managerial Sciences

There is a need to better understand how groups of people collaborate in large projects involving a variety of participants sharing a rich but uneven distribution of knowledge and imagination among them. Software engineering research would be enhanced by greater interaction with behavioral, managerial, and other scientists that could lead to increasingly effective contributions to software engineering practice, in part by accelerating the transfer of technology into and through the software engineering community. The field has benefited

in the past from technology transfer: for example, configuration management practices and change-control techniques developed in the aircraft industry were adopted in the 1950s and 1960s.

There may be particular value in augmenting the insights of computer science and electrical engineering with the insights of behavioral and managerial sciences. Since large software systems will continue to be produced by teams for the foreseeable future, insights gained in other team contexts may be useful for software engineering. To get those insights it may be necessary for software engineers to actually team up with specialists from other disciplines; the benefits of such cross-disciplinary teams have been demonstrated, for example, in the area of ergonomics, where cognitive and management science specialists have been brought in to determine how best to complement human skills with automation. Even within computer science, some areas other than software engineering have aging software platforms that need to be reimplemented to make them less brittle and more easily changed or to improve the user interface to take advantage of workstation technology advances. In such areas software engineers could collaborate with other types of computer scientists and engineers in new developments that both produce new tools and serve as the objects of study. The CSTB workshop pointed to a need for software engineers to glean insight from people with complementary expertise but did not develop the concept.

Develop Additional Directions

Software engineering research today follows a variety of patterns, including the following:

- building systems with certain properties to show their feasibility;
- measuring properties of one or several systems;
- improving the performance of systems along particular dimensions;
- developing abstract formal models for certain domains;
- showing how to describe phenomena by designing languages; and
- making incremental improvements on prior work.

All of these activities are relevant to complex software systems. But given the nature of those systems and the problems we face today, some new approaches to research may also be productive.

CSTB workshop participants recommended that the academic research community expand its notion of good research to accept review or synthesis studies, case studies, comparative analyses, and development of unifying models for individual or multiple domains. In particular, review or synthesis studies, which are common in a number of other fields, would support a greater and ongoing codification of software engineering knowledge and help to minimize the reinvention of techniques and processes. Finally, if effective handbooks are to be developed, as recommended above, research that supports such handbooks must be encouraged and rewarded.

CONCLUSIONS

Modern society is increasingly dependent on large, complex computer-based systems and therefore on the software that drives them. In many cases, systems designed 20 years ago still provide a foundation for large businesses—and such systems are becoming unmaintainable. As the ability to manipulate, analyze, and grasp information has been magnified by information systems, so also has the appetite to process more and more information. Each new application has generated ever more complex sets of software systems. In the past few years, problems with such systems have cost millions of dollars, time, and even lives in applications ranging from aviation to controls for medical devices. Improving the quality and the trustworthiness of software systems is a national priority, but it is also a problem that seems forever to receive less attention than it deserves because software systems seem invisible, are poorly understood by laymen, and are not even adequately addressed in universities. Managers are consistently surprised by the inability of software engineers to deliver software on time, within budget, and with expected functionality. The nation should not have to wait for a catastrophe before it tries to enhance this critical resource.

The software research community has ridden the waves of several advances; expert systems and object-oriented programming have been among the topical foci of researchers during this decade. Although large-system developers benefit from these and other advances, the software systems challenge is fundamental and is not amenable to solution through single categories of advances.

As discussed in this report, a necessary first step is for the software engineering community and managers who procure and use large software systems to adopt a more realistic vision of the complex software system development process. Following on, the direction and conduct of software engineering research should be both broadened (in particular, by fostering interactions with practitioners) and made more systematic, through codification and dissemination of knowledge as well as an infusion of more mathematics, science, and engineering. Good problems make good science and engineering—and good problems in the software development community are being bypassed because software engineering researchers are unable to deal with them in a structured, rigorous manner.

The Chinese pictograph for "crisis" is composed of the characters for "danger" and "opportunity." The wisdom this represents is worth noting as we grapple with the looming crisis in our ability to create and maintain large and complex software systems. The danger is that soon we might not be able to create the software our business and government applications need. The opportunity is there for the software engineering research

community to find new and fruitful directions in the problems faced by practitioners.

REFERENCES

1. Ad Hoc Committee on the High Cost and Risk of Mission-Critical Software. Report of the USAF Scientific Advisory Board Ad Hoc Committee on the High Cost and Risk of Mission-Critical Software, US Air Force Scientific Advisory Board, Dec. 1983.
2. Association for Computing Machinery. *Proceedings of Conference on Computer-Supported Cooperative Work* (Sept. 26–28, 1988, Portland, Ore.). ACM, N.Y.
3. Balzer, R. A 15 Year Perspective on Automatic Programming. *IEEE Transactions on Software Engineering 11*, 11 (1985), 1257–1268.
4. Belady, L. Software is the glue in large systems. *IEEE Communications Magazine*, Aug. 1989.
5. Belady, L., and Lehman, M.*Program Evolution Processes of Software Change*. 1985. Academic Press Ltd., London, UK.
6. Bernstein, L., and Yuhas, C. The chain of command. *UNIX Review*, Nov. 1987.
7. Besemer, D.J., et al. A Synergy of Industrial and Academic Education. Technical Information Series, General Electric Corporate Research and Development, Schenectady, N.Y., (Aug. 1986).
8. Biggerstaff, T., and Perlis, A. *Software Reusability: Concepts and Models 1* and *Software Reusability: Applications and Experience 2*. Addison Wesley/ACM Press Frontiers in Science, N.Y.
9. Boehm, B.W. *Software Engineering Economics*. 1981. Prentice-Hall, Englewood Cliffs, N.J.
10. Boehm, B.W. A spiral model of software development and maintenance. *IEEE Computer 21*, 5 (May 1988), 61–72.
11. Booch, G. *Software Components with Ada*. 1987. Benjamin/Cummings Publishing, Menlo Park, Calif., 571.
12. Bowen, T., Wigle, G., and Tsai, J. *Specification of Software Quality Attributes*. RADC-TR-85-37, *1*, Rome Air Development Center, 1985.
13. Brooks, F.P. No silver bullet—Essence and accidents of software engineering. IFIP's *Information Processing 86*, H.J. Kugler (ed.), 1986 Elsevier Science Publishers B.V. (North-Holland).
14. Cleaveland, J.C. Building application generators. *IEEE Software*, July 25, 33, 1988.
15. Cody, W.J., and Waite, W.M. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, N.J., 1980, 269.
16. Computer Science and Technology Board, National Research Council. *Toward a National Research Network*. National Academy Press, Washington, D.C., 1988.
17. Curtis, B., Krasner, H., and Iscoe, N. A Field Study of the Software Design Process. *Commun. ACM 31* 11 (Nov. 1988), 1268–1287.
18. Defense Science Board Task Force on Military Software. *Report of the Defense Science Board Task Force on Military Software*, Office of the Under Secretary of Defense for Acquisition, Sept. 1987.
19. Freeman, P. A conceptual analysis of the draco approach to constructing software systems. *IEEE Transactions on Software Engineering* (July 1987), 830–844.
20. Guindon, R. (ed.). *Cognitive Science and Its Applications for Human-Computer Interaction*. Lawrence Erlbaum Assoc., Hillsdale, N.J., 1988.
21. Knuth, D. *The Art of Computer Programming*. Addison-Wesley Series in Computer Science and Information Processing 2 (1973). Addison-Wesley, Reading, Mass.
22. Martin, R.J., and Osborne, W.M. Guidance on software maintenance. NBS Special Publication 500-106. Computer Science and Technology, U.S. Dept. of Commerce, National Bureau of Standards, Washington, D.C., Dec. 1983, p. 6.
23. McCabe, T. Complexity Measure. *IEEE Trans. on Software Eng.*, 2, 4, 308–320.
24. Microelectronics and Computer Technology Corporation, Conference Committee for CSCW '86. *Proceedings of Conference on Computer-Supported Cooperative Work* (December 3–5, 1986, Austin, Tex.). MCC.
25. Mills, H.D. Benefits of Rigorous Methods of Software Engineering in DoD Software Acquisitions, Feb. 1989.
26. Mosemann, L.K. Software Engineering and Beyond: The People Problem. Keynote address at the SEI Affiliates Symposium, Software Engineering Institute, Carnegie Mellon Univ., (May 2–4, 1989, Pittsburgh, Pa.).
27. Neighbors, J.M. The draco approach to constructing software from reusable components." *IEEE Trans. on Software Engineering 10*, 5 (1984), 564–573.
28. Partsch, H. and Steinbruggen, R. Program transformation systems. *ACM Computing Surveys 15*, 3 (1983), 199–236.
29. Perry, R.H., et al. *Perry's Chemical Engineers' Handbook*, (1984). McGraw Hill, N.Y., 1984.
30. Royce, W. Managing the development of large software systems. *Proceedings WESCON*, (August 1970).
31. Shaw, M. Beyond programming-in-the-large: The next challenges for software engineering. Technical Memorandum SEI-86-TM-6, Software Engineering Institute, Carnegie Mellon Univ., Pittsburgh, Pa., May 1986.
32. Shaw, M. Larger scale systems require higher-level abstractions. *Proceedings of the Fifth International Workshop on Software Specification and Design*, (May 19–20, 1989, Pittsburgh, Pa.). ACM, N.Y.
33. Smith, D., Kotik, G., and Westfold, S. Research on knowledge-based software environments at Kestrel Institute. *IEEE Trans. on Software Engineering 11*, 11 (1985), 1278–1295.
34. Swanson, E.B., and Lientz, B. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. 1980. Addison-Wesley, Reading, Mass.
35. Williams, B.G., Mui, C.K., Johnson, B.B., and Alagappan, V. Software design issues: A very large information systems perspective. Center for Strategic Technology Research, Arthur Andersen & Co., Chicago, Sept. 28, 1988.

For a complete copy of this 92-page report, contact National Academy Press, 2102 Constitution Ave., Washington, D.C. 20418.