Final Report
NSF Workshop on a Software Research Program For the 21st Century
Greenbelt, Maryland, October 15–16, 1998

Workshop Participants:
Professor Victor R. Basili, University of Maryland (Chairman)
Mr. Laszlo Belady, Belady Enterprises
Professor Barry Boehm, University of Southern California
Professor Frederick Brooks, University of North Carolina
Professor James Browne, University of Texas
Dr. Richard DeMillo, Bellcore
Dr. Stuart I. Feldman, IBM
Dr. Cordell Green, Kestrel Institute
Dr. Butler Lampson, Microsoft Corporation
Professor Duncan Lawrie, University of Illinois
Professor Nancy Leveson, Massachusetts Institute of Technology
Professor Nancy Lynch, Massachusetts Institute of Technology
Dr. Mark Weiser, Xerox Corporation
Professor Jeannette Wing, Carnegie Mellon Institute

Executive Summary

In August 1998 the President's Information Technology Advisory Committee (PITAC) submitted an Interim Report emphasizing the importance of software to the nation and calling for a significant new federal investment in software research.[1] An NSF workshop subsequently brought together representatives of a broad segment of the software community to discuss the software research agenda. Workshop participants included researchers and developers from geographically diverse organizations in academia and industry.

A major theme of the PITAC Report was the "fragility" of our software infrastructure, where fragility means "unreliability, lack of security, performance lapses, errors, and difficulty in upgrading." The PITAC was particularly concerned by these failings, because software now affects almost every aspect of personal and professional life in the nation. It manages our telephone networks and nuclear power plants; a large variety of embedded control and sensor devices, air-traffic-control systems, and the readiness of the world's most advanced military force, to mention only a few examples. Given the exponential growth curve for software use, we expect even greater demands on software in the future.

To meet them, the workshop concluded that software research has to expand the scientific and engineering basis for constructing "no-surprise" software of all types. We 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 test-beds for the new ideas and address a serious problem identified by the PITAC: "desperately needed software is not being developed."

A significant new research investment is required to understand and correct the software "fragility" problem. The following workshop report discusses the issues in more detail. Below is a summary of the detailed findings and recommendations.

FINDINGS AND RECOMMENDATIONS

F1 Current software has too many surprises. The sources of surprise are poorly understood.

R1 Emphasize empirical research aimed at understanding the sources of software surprises.

F2 Key sources of software surprise include immature or poorly integrated software domain sciences, construction (product) principles, and engineering processes. Software research emphases have swung from process to product re-

[1] See: http://www.hpcc.gov/ac/

search, with weak coverage of domain sciences and integration.

R2 Balance and incrementally expand research in the domain sciences, construction (product) principles, engineering processes, and their integration.

F3 Key components of software surprises include scalability, evolvability, dependability, usability, performance, and predictability of cost and schedule.

R3 Emphasize the ability to address these issues in research support and evaluation.

F4 Software technology is hard to transition into practice and feedback is needed on its effectiveness. Just doing software research is not enough.

R4 Expand initiatives to transition research and provide feedback to researchers via government/industry/academic collaboration, incentives, and support.

1. Introduction and Motivation.

As an industry, information technology, especially software technology, has had an immense impact on the U. S. economy. In his MIT commencement address on June 5, 1998, President Clinton observed that "in just the past four years, information technology has been responsible for more than a third of our economic expansion." Important new software technologies with the potential for driving the economy to even greater heights—electronic commerce and advanced communications networks, to mention only two exciting examples— are now under development or already in the field.

The software story is not one of unvarnished success, however. In their Interim Report to President Clinton, the President's Information Technology Advisory Committee (PITAC) calls software "the new physical infrastructure of the information age ... fundamental to economic success, scientific and technical research, and national security" but observes that "the Nation currently depends on software that is fragile, unreliable, and extremely difficult and labor-intensive to develop, test, and evolve." The PITAC Report notes the increasing importance of software "for commerce, for communication, for access to information, and for the physical infrastructure of the country." But it also warns that "our ability to construct ... needed software systems and our ability to analyze and predict the performance of the enormously complex software systems that lie at the core of our economy are painfully inadequate. We are neither training enough professionals to supply the needed software, nor adequately improving the efficiency and quality of our construction methods."

The NSF Workshop on a Software Research Program for the 21st Century, which was held in Greenbelt, Maryland, on October 15–16, 1998, examined and elaborated the PITAC recommendations for significant new research efforts towards understanding how to construct, analyze, and evolve software. The discussions ranged over all types of software from the everyday variety that dominates the commercial activities of the nation to the most complex, leading-edge software. Workshop participants believe that the best approach to improving software quality and software engineering productivity begins with understanding and building on the substantial successes of the last twenty-five years of software development. In the sequel, we will discuss research strategies for:

- extracting useful principles of software construction through empirical investigations of successful projects and validating design principles developed in the research literature and elsewhere;
- advancing our understanding of the software engineering process by experimenting with new approaches in applications projects;
- continuing to develop a rigorous formal basis for software development that is sensitive to issues raised by empirical analysis and that puts a special focus on adapting to change and scaling to systems of realistic size;
- forming teams to build important advanced applications that will both serve as a testbed for the new ideas and help address a problem identified by the PITAC—that "desperately needed software is not being developed;" and
- Emphasizing human factors for both software products and processes.

2. Defining the Basis For a Software Discipline.

The PITAC Report spotlights the fact that "the software our sys tems depend on is fragile" and notes that the "fragility is mani fested as unreliability, lack of security, performance lapses, error and difficulty in upgrading." In other words, too much softwar has too many surprises. Our goal should be to develop technique for expanding the envelope of "no-surprise software" and for ur derstanding more precisely when we are in danger of crossing th surprise threshold. Then, if we need an application outside thi threshold, it will not be a surprise if it experiences overruns c shortfalls.

The no-surprise software envelope. We should keep in mind, how ever, that a large amount of no-surprise software is created eac year by organizations that have developed a standard engineerin approach from long experience in stable domains. The develop ment of such business systems as payroll and order-entry is han dled well by many experienced companies as long as the rules c legalities governing the systems have not changed radically an the computational environment is well understood. One importan characteristic of such systems is often the existence of some dom nating technology—a centralized database, for example—or regu lations that impose constraints on the construction of system Numerous companies use development methodologies that, whi not the mathematically rigorous formal methods of the researc community, are nevertheless systematic engineering approache As in classical engineering, they produce appropriately functiona reliable, and maintainable systems, usually on time and withi budget. Other examples of this success include various kinds manufacturing process-control systems, many NASA satelli ground-support systems, and even some kinds of air-traffic contr systems.

Software development is similar to other engineering activities: v reach our engineering limits whenever the environment for a n surprise system changes significantly. Sometimes technology governing regulations change, or it may be that the system mu tolerate dramatically increased stress. Lacking processes to filt commitments to unachievable success levels, we can easily exce our engineering capabilities. One common example is the busine

system that must handle an order-of-magnitude more transactions with many interdependencies than similar previous systems. The same thing happens if we greatly increase the size and complexity of a database support system—for example, the ground data Earth Observing System of NASA, which has been under development for more than a decade.

Classical engineering disciplines understand the surprise/no-surprise threshold better than we do. They know how to limit change, calculate its impact, and, over time, make a systematic transition to increased capability. We need careful empirical studies of the methods used by the developers of successful no-surprise systems, with a view to identifying and generalizing their methods. We also have to investigate projects that failed when they exceeded the threshold of current engineering knowledge and learn how to recognize the threshold and systematically push it higher. One goal should be to develop ways to bound the development problem and constrain its solution and to understand and exploit the relationship between the two.

We believe that the software research community would profit greatly by having a deeper understanding and appreciation of the large number of successful no-surprise systems built every year by the software industry. In our view, this is the proper starting point for addressing the problems of the many important software systems that are built beyond the threshold of current engineering practice.

These latter "outside-the-envelope" systems are of great interest to us as well. They see wide use and may often work satisfactorily, but their development and maintenance costs can be very high, and, in many cases, they have unacceptable failure levels. Outside-the-envelope systems include telephone switching systems and local area networks. Finally, there are the unsuccessful and sometimes highly visible systems that go well past the threshold of successful engineering practice—e.g., the FAA Advanced Automation System. A primary question facing the software discipline is: How do we learn from building no-surprise systems and apply that success to outside-the-envelope and unsuccessful systems, as complexity and risks grow?

Software-related research areas. For explanatory purposes, let's classify software-related research into three areas: domain science, the principles of construction, and the engineering process. This taxonomy can shed light on the key issues in any engineering discipline. For example, in civil engineering: 1) the domain science is that part of physics, materials science, engineering economics, engineering ergonomics, etc., that is useful for building bridges, roads, and other relevant artifacts; 2) the principles of construction are those scalable general principles for creating civil engineering objects; they would permit the construction of useful prototypes; and 3) the engineering process is the standard practice that enables a well-trained civil engineer to build a real bridge with available materials and construction crews of normal skill, and operating under realistic time constraints and budgets.

In software engineering, domain sciences for applications include computer science plus physics, accounting, and so on; for operating systems, it's computer science. Except at the (usually non-scalable) level of algorithms, large development teams generally use only non-formal, non-validated principles of construction—for example, the commercial methodologies for building routine systems. Research into the science of software construction often does not address scalability, except to acknowledge that it is an issue. We have very rigorous underpinnings for certain foundational areas of software—formal verification, formal specification of designs or requirements, for example—but much less for the principles of construction. (Program synthesis from high-level specifications is an important exception.) Research results in foundations are seldom accompanied by equally rigorous and usable techniques for constructing real-world software based on the formal representations and underlying theory. Lacking a sufficient construction science, it has been difficult to create a realistic, rigorous software engineering process.

Why is this important? Because the software problem is an engineering problem. As with all engineering disciplines, software requires rigor in the underlying sciences:

- *Domain science.* We need advances in the science of computing, because it is an essential domain science for software engineering. The study of algorithms and their characteristics will continue to be an important foundation for software. We should pay special attention to scalable properties that can usefully be isolated at the algorithmic level—for example, properties of models of system interconnections.
- *Construction principles.* As Mead and Conway wrote twenty years ago in establishing foundational principles for VLSI design: "The task of designing very complex systems involves managing, in some highly structured way, the space and time relationships between the various levels of system building blocks so that the entire system will function as intended when it is finished." Commercial practice advocates many principles for "managing ... relationships between the various levels of system building blocks," and the software research literature is replete with design principles, from the highest to the lowest levels. In virtually all cases, the principles have neither been carefully classified for applicability nor validated for practical effectiveness. We believe those scientific activities should have high priority in future research projects.
- *Engineering process.* Much research has been conducted on the software engineering process in recent years, and our understanding of the importance of process has increased. The stage is now set for developing a customizable standard engineering practice founded on a sophisticated set of validated tools and techniques from a relevant science of construction principles. This practice should also draw heavily on empirical investigations of successful no-surprise systems from various important domains.

This breakdown gives us a more careful way of talking about the "science" of software. In particular, it is now easier to address a question that occasionally arises: Is it possible that there is no science of software? That is, could building high-quality software be more a matter of artistic skill and good taste than a scientifically well grounded activity? Within our framework, an appropriate reply might be: Which science do you mean?

Domain science clearly exists for constructing the software that compiles a program written in a conventional programming language for a conventional computing platform. But that science is far from enough to cover the computing and human aspects of a large, unprecedented air-traffic control system subject to simultaneous safety, evolvability, and resource constraints. To provide a

scientific basis for developing and evolving such systems will require the extension of the domain science—computer science, in this case—to provide capabilities for large-scale, distributed, ultra-reliable, real-time information capture, processing, management, and display. It will also require the integration of computer science with other domain sciences, such as the aero-sciences, economics, and social sciences to address such issues as collision avoidance, human-computer interaction, computer-supported collaborative work, and risk management. Thus, to more rapidly bring ambitious, outside-the-envelope software systems within the scope of no-surprise development techniques will require improvements not only in computer-domain science, but also in the collaboration of computer science with other domain sciences. Doing all this may sound unachievable, but as a benchmark, there exist medium-size, air-traffic control systems that have passed the no-surprise test. One example is the system developed by Raytheon for Norway. Such benchmarks indicate that for some complex but well-understood applications, there is a sufficient combination of specialized subsets of various domain sciences (in addition to appropriate construction principles and engineering processes, of course) for experienced organizations to produce no-surprise software systems. The major challenge is to extend this engineering capability so that we can address more ambitious, unprecedented systems, which are exactly those needed for future industrial competitiveness and future public services to enhance our quality of life.

As we examine the sciences underlying software, the existence of domain science is indisputable. But what about construction? Construction principles are implicit in the development of successful no-surprise systems and explicit in methods long discussed and advocated by the research community (not to mention the community of commercial design consultants). Therefore, the appropriate questions are: Do the principles rest on a coherent scientific foundation? Are they actually useful for building software systems?

The first question is easy to address. Researchers have frequently recommended this or that design principle after developing (or outlining) sound mathematical foundations. Design by abstract data types is an obvious example, to which we might add stepwise refinement, structured programming, and decomposition of multi-party interactions, to mention only a few popular examples. Sometimes, the principles are at a much higher level— Dijkstra's statement that distributed systems should be designed as self-stabilizing systems, for example—but equally well founded in appropriate mathematical theory. There are a very large number of instances of mathematically well-founded construction principles.

The second question—usefulness—is clearly addressable in principle, but the fact that it is rarely attempted continues to separate software from other engineering fields. The usefulness of our design principles and, therefore, of the underlying science is a matter for empirical investigation, and that has not been a primary focus of software research.

Finally, consider the software engineering process. It also has a validation problem, as well as a foundational issue—our process is rarely based on carefully stated construction principles—but the "science" underlying it is akin to the same management science that supports all other engineering disciplines.

Findings:

- Software research must address the wide range of software needs: from systems we can build reasonably well (no-surprise systems), to systems that test our ingenuity, to unprecedented systems that are beyond our current abilities.
- Research is needed in the underlying domain sciences, construction principles, and engineering processes.

Recommendations:

- We need to understand the state of the art and the state of the practice of software development when we try to define software research needs. For example, we have to answer such questions as, "What are the major difficulties in moving from no-surprise systems to unprecedented systems?" To do so wil require a greater focus on the empirical study of existing systems.
- Software research requires advances in the underlying domai sciences (including computer science), the principles of software construction, and the engineering process. We need to create rigorous principles for software development, apply these principles to various classes of unprecedented systems, develop support methods and tools for using them, and undertake controlled studies of their application for continued learning about software development and evolving the principles.

3. One Example of an Important New Application.

Past successes demonstrate that some software development o ganizations can build no-surprise software until conditions chang too much, and a discontinuity occurs between capability and e pectation. The problem can arise because of a deficiency in any the three areas discussed above—domain science, constructio principles, or engineering process—but it usually comes from th latter two. As we try to understand development failures, we nee to identify where, within this or some other systematic framewor the breakdown occurred and what new knowledge will be require to push beyond it in the future. This systematic study of develo ment problems characterizes classical engineering fields and hel explain their steady progress. As Levy and Salvadori put it in *W Buildings Fall Down,* engineers "learn a lot from failures," typ cally through systematic investigation by experts.

The software field requires special help to make the transition engineering, because we are driven much more than most fields t the rapid pace of our market. The industry is motivated primari by the need to meet market demands and cannot slow down study principles and process. But when the demands on systen are growing exponentially, we cannot afford to let engineeri knowledge grow linearly.

To illustrate likely demands on 21st–century software and t evolution of the demands that move a system out of the "n surprise" envelope, we provide a description of the evolution of sample application: electronic commerce.

Electronic Commerce. A few years ago, electronic commerce w virtually non-existent. Now it has grown to at least $6,000,000,000 business, and predictions for growth are extreme optimistic. Estimates of total worldwide E-Commerce from rep table information technology analysts include:

Year	2001	2002	2005
$Amount	$200B	$400B	$1000B

Over time, these estimates have continually increased.

The most visible part of this activity, involving individual consumers, is only about 20% of the dollar volume, though a much higher percentage of the total number of transactions. As many as forty-million people have bought or sold something on the Web. This number will easily rise past one-hundred million in a few years, and we can expect one billion customers worldwide by 2010.

Electronic technologies decrease the cost of many important economic activities by (literally) orders of magnitude. Historically, any large change in the factors or production and distribution eventually leads to huge dislocations and opportunities. Luckily, the United States is in the forefront of this technology and is in a position to take early advantage. The current successes in electronic commerce are directly attributable to the work of computer scientists over the past couple of decades. We would not be able to shop on the Web—to locate goods there and pay securely for them—without important work in the underlying domain sciences such as cryptography, payment protocols, the Web itself, the Internet, large-scale distributed systems, distributed database management systems, and human-interface interaction models.

Investments in these research areas have been repaid many-fold by the growth in the American economy. (Just consider the market capitalization of Yahoo!, Amazon, eBay and similar companies.) And there are many bright expectations for the future, moving from occasional catalogue sales to a standard mode of doing business—perhaps the only mode in certain new areas. This move will be a genuine change in economic organization around the world and will change behaviors of the average citizen. There will be major dislocations and discomforts, but the net result will be a much more efficient and (we hope) more equitable economy.

Many problems will need to be solved to reach this desirable state. As e-commerce moves from novelty to business necessity, our tolerance for discomfort and failure will disappear, and the need to support transactions from anywhere at anytime without losing orders or payments will be essential. The e-commerce world will have to work with telephone-system reliability. The market will tolerate essentially no regional or global outages and very few local problems. Yet we will continue to have fallible networks, computers, peripherals, and software. Commerce will be conducted with firms around the globe, products will be ordered at all times, and new services will be created at an accelerating rate. Somehow we must be able to provide improving service in many styles and languages over the world's largest, most complex distributed computer system. Furthermore, there will be very strong security requirements, to ensure that the parties to any transaction are identifiable and have appropriate authority (financial, organizational, political). The Web makes possible an entirely new level of privacy invasion and intrusion. It is possible to track every move on the Web, including time and even physical location. On the other hand, it is also possible to hide one's identity in ways that are impossible in the traditional physical world. The tension between these, and the risks of social problems on the one hand and economic ones on the other, creates a need for much research, applied research, and advanced development.

Most business today is transacted on the basis of fixed prices, standard goods, and an agreement between a single buyer and seller. In the future, much more complex interactions may become the norm, involving more parties (for comparison shopping, competitive bids, alternatives, bundling, etc.). Traditional styles of database transactions may be woefully inadequate to support the new opportunities, about which we can only guess. There will be a rising level of experimentation in this world, so applications will be designed, tried, redesigned, in huge numbers and on a very short time scale. The set of applications and services that a user sees will, therefore, change from instant to instant, and many will be faulty, yet the overall economic and computing system must not falter.

As e-commerce becomes more common, we will move from a world with 10^7 occasional participants to one with 10^9 frequent users and with enterprises doing very large fractions of their buying and selling by the new means. Consumers will have many ways to communicate and at least 10^{11} software agents and 10^{11} network-enabled gadgets. The new modes of interaction and new ways to control problems and the need to provide continuous availability pose truly grand challenges. In such a world, it is not obvious how to allocate resources appropriately or how to charge for them—economic approaches with competing agents and bidding offer one of the few plausible solutions. Finally, the software will not all be written by specialist professionals but in many cases will be provided in hosted environments with standard tools, or using software kits. How do we make it easy yet foolproof to specify and implement complex business models?

Within the framework of this report, other questions naturally arise: Where are the principles of construction that apply to building such systems? What is the engineering practice available? How do we determine which of the ever-expanding set of requirements will force us out of the envelope to systems that we cannot build reliably?

Findings:

- Software systems are evolving at an enormous rate of complexity, driven by the needs of a very active marketplace.
- The example application demonstrates the limits of our existing domain models, principles of construction, and engineering processes.
- At the boundaries of no-surprise systems are the significant issues of scalability, system evolution, and engineering process.

Recommendations:

- We need progress in both the underlying software domain science (e.g., databases, security, human-computer interaction, distributed systems) and the new views of the application domain science (e.g., e-commerce).
- Research is required on the principles of construction and the engineering process for the development of unprecedented software systems, including problems in scalability (e.g., increases in the user set, the amount of functionality), evolution of systems (e.g., need for almost instant change of capabilities), process/product relationships (e.g., the need to predict and achieve ever-higher levels of reliability, availability, and performance; the need for tool support).

4. Recommended Directions.

The PITAC Interim Report recommends substantial new investment in basic information technology research, noting that "the NSF defines basic research as the study of the 'fundamental aspects of phenomena and of observable facts without specific application toward processes or products.'" Applied research, on the other hand, "is aimed at determining the means to meet specific needs," while development is "the systematic use of knowledge to produce useful materials, devices, or methods." The Report observes that R&D is much more than these definitions. It is "a complex non-linear interaction between concepts and theories, data and experiments, and new products and processes." The concepts, theories, data, and experimentation produced by "basic research is a critically important part of this interwoven system."

We agree and would emphasize that both applied research directed at particular application domains and the development of particular products or services inevitably require a foundation of basic research, which solves problems that form the barriers to real progress and often can be applied across domain boundaries. Thus, a solution to the problem of scalability, one of the most important basic research questions for software, would have a wide impact over many important application domains. To take another example: research that improved the software engineering process would positively benefit the development of almost all software.

In well-established engineering disciplines, basic research can focus primarily, but, perhaps, not exclusively, on fundamental technical issues. In a field as young and dynamic as software, however, the research community must also put substantial effort into establishing the principles and basic components of the discipline. The software research community has been doing that by asking such questions as: How do we obtain observable facts? What are the fundamental variables of the software discipline? But the answers have been slow in coming, because the same forces that have made information technology "responsible for more than a third of our economic expansion" over the last four years—as President Clinton told the MIT commencement audience last June—have also put a focus on applied research to support new applications, rather than a fundamental understanding that could eventually lead to extraordinary increases in our ability to produce no-surprise software. In this section we argue for supporting *both* high-risk fundamental research and the exciting new technologies that come out of applied research.

4.a. Problem areas.

To illustrate the need for greater research investment in the software discipline, let's look at three significant basic research technical problem areas.

1. Scale-up. One of the most important expansions of the no-surprise envelope will come when we have a coherent and general approach to scalability. In various domains, we know how to build no-surprise systems of a certain size and complexity, but the scalability question asks, "How do we *systematically* expand our ability to create more complex systems for a given domain?"

One reasonably successful approach to scaling up construction has been to build large reusable components: operating systems, Web browsers, databases, accounting systems, and office productivity systems are all examples. These are very large components, often containing millions of lines of code, and they make most applications much easier to build than in the past.

We should investigate why the approach has worked with very large components, while smaller-scale reusable software has not been nearly as successful. In particular, we need to understand why certain attempts to capitalize on these ideas have not been successful—e.g., uses of COTS where the assumption was that we would get over 70% reuse and ended up with at most 30%. Are the failures traceable to missing principles of construction using components, do they arise from an inadequate or inadequately followed software engineering process, or do they arise from a lack of understanding of the real system requirements? How do we systematically create a component-based development methodology that builds on the large-scale successes?

2. Evolving systems. Understanding how to expand the no-surprise envelope for systems subject to continual evolution is extremely important, because evolvability is a widespread requirement. Software systems enable us to improve the way we do business. They also accelerate changes in the way we do things, which causes our requirements for software solutions, and thus the software itself, to change along with them.

This increased rate of change comes from a variety of factors. For example, as a particular business domain changes, the software must continue to adapt to be relevant. But there are also changes that are driven by the nature of the software itself. As users become more knowledgeable about a system, they understand their needs better and, hence, have new requirements. In addition, their expectations change.

As the previously described example of e-systems demonstrates, requirements for certain kinds of important systems will continuously evolve. The more we use such systems, the larger and broader the user community will become, and the more rapidly the requirements will grow.

A significant number of large systems in the future will, by their nature, continually evolve. Changes will expand from component changes to architectural changes, from manual changes to automated changes, and from off-line changes to on-line changes (where the change occurs while the system is running).

How do we systematically develop principles of construction and software process appropriate to the needs of evolving systems?

3. Process/product relationships. Finally, we need to expand the no-surprise envelope by developing principles and processes for producing software that has specified characteristics. That is, given a particular set of system characteristics, how do we systematically determine the principles of construction and the software engineering process to build a system with those characteristics? As in most disciplines, we need to understand the cause-effect relationship between various processes and how they affect or make possible various product characteristics. Even though human factors play a large role in the software discipline, it is still important to understand how issues like risk and predictability are effected by how we choose to build the system.

We currently have trouble answering such questions as: What are the most appropriate processes, techniques, and tools for effectively constructing or analyzing particular classes of software products? What are the levels of specificity for defining such processes so they support, but do not constrain, the development

evolution of a system? How do we predict such aspects as the cost, delivery time, and reliability of large systems given certain variables known or estimable in advance? How do we balance people factors: individual vs. group incentives; autonomy and empowerment vs. disciplined, uniform processes?

We need to build models of cost, reliability, and effectiveness based on observations of real projects. Creating these models will require empirical investigation as well as model building techniques that are sensitive to the needs of the software discipline.

4.b. Building a software discipline.

Why do we need a software discipline? Classical engineering disciplines produce reliable artifacts under schedule, budget, workforce, legal, and other real-world constraints because they have a consistent framework—their scientific foundations—for developing and communicating standard practice and for analyzing and correcting faulty practice. Within this framework, academic departments educate the next generation of practitioners, who can then provide important information on real-world problems using the same framework. Properly educated engineers are also capable of profiting from new engineering research. By contrast, few among the hundreds of thousands of working software engineers have an academic background in the design of complex systems. Thus, few engage in a dialogue with the research community.

Without question, a coherent scientific framework has proven to have great value for classical engineering. Bringing to maturity the scientific foundations of software would, therefore, appear to be highly desirable, although some worry that trying to impose engineering discipline on the field would risk losing the flexibility that has made software one of the driving forces of our economy. They are concerned about dampening the creativity that has produced some of the most impressive technology of the Twentieth Century.

We should keep in mind, however, that adherence to a rigorous development process has not prevented classical engineering from undertaking high-risk, high-payoff projects. Further, the validity of Moore's Law for electrical engineering over three decades demonstrates that a field can be both rapidly developing and economically significant while operating under engineering constraints. Even if some software will always be built without a rigorous engineering approach—because of market forces, for example—it will be important to maximize the no-surprise core of such systems and to quantify the risks of going outside the no-surprise envelope. This knowledge of where you are in the no-surprise-to-high-risk continuum is characteristic of classical engineering but not of software engineering as currently practiced.

Without a scientific basis for the software discipline, we cannot build no-surprise systems of the next magnitude. We will remain prisoners of fads, rather than participants in the engineering enterprise, if we lack a sufficient basis for carefully choosing the right construction principles and engineering processes. Good solutions for systems that should fall within the no-surprise envelope will continue to be derivable only by an ever-decreasing number of gifted designers, who should, instead, reserve their efforts for the most difficult and advanced systems. Each success will itself be a non-repeatable surprise, and there will many more failures than successes.

We agree strongly with the PITAC recommendation for significant new investment in basic software research. Given the large number of no-surprise systems from which researchers can now draw important general principles and the demand for extraordinarily complex and dynamically evolving new systems, the time is right for creating better scientific foundations for the field.

Recommendations for software research. Understanding a discipline involves observation, reflection, encapsulation of knowledge, the creation of evolving models (of both application domains and problem-solving processes), and experimentation. This paradigm has been used in many fields—e.g., physics, medicine, and manufacturing. The differences among the fields are reflected in how models are built and analyzed and how experimentation is performed.

Currently, there is an insufficient set of models to support reasoning about the software discipline, a lack of understanding of their limitations, and insufficient analysis and experimentation within a model-driven framework. Thus, the software discipline needs to bring appropriate research methods to maturity and evolve them as we learn and grow as a discipline. In particular, we need to study and classify past development successes and failures relative to the parameters that limit our progress. We also need to build bodies of knowledge by classifying and integrating research results.

Too often, promising software research goes unevaluated. Lacking a proper understanding of the usability of an idea, overloaded, risk-averse applications organizations rarely pay attention to research results. This lack of engagement means that the research community does not receive feedback on the viability of new approaches.

If we focus on research to produce no-surprise software, one thing we would want is no-surprise research papers. These are papers in which the research results are accompanied by the author's best effort to determine the region of successful applicability of the results. For example, a research paper on a static code analysis technique would not just furnish the technique and an example of how well the technique worked on a toy program. In addition, it would include an analysis of how well the technique worked on various classes of code (e.g., very efficient for single modules; very inefficient for multiple modules with multiple threads of control). The effort required for such analyses should be enabled by, and expected from, the award of larger research grants.

This information makes it much more likely that applications organizations will experiment with them. It also provides a context for the experimental results to refine the assessment (e.g.: strong for 20-statement modules; weak for 2000-satement modules).

Not only do these assessment results help other applications organizations benefit from the capabilities, but they also help sharpen the future research agenda, as researchers discover which capabilities are weakly covered and in high demand.

This feedback-and-improvement loop can be both strengthened and accelerated if the applications organizations are motivated to collaborate with the researchers in performing the evaluations. For example, if a PITAC software initiative provided matching funds for university-industry collaborations to evaluate and strengthen the research results, the assessments would provide timely benefits to the adopter, stronger results for the researcher, and better research and technology status information for the remainder of both the research and the application communities.

As we have pointed out, the software community has achieved greater and greater success in support of other disciplines at the cost of not investing in and evolving the scientific and engineering basis for the software discipline. Putting so much effort into applications for others explains, in part, why we have rarely studied even those systems that represent our successes. We have not developed a significant understanding of how we have bound a problem space (for example, by concentrating on a particular application domain or by minimizing the degrees of freedom of the requirements) or constrained a solution space (for example, by using standard development methods or running on a well-understood computer system). We do not have a careful approach to analyzing the effect of changing a few parameters in system requirements. For example, we will not be able to systematically extend the no-surprise envelope until we can precisely describe why an organization that has been building order-entry systems for 10,000 orders encounters high risk when trying to construct the same kind of system for 2-million orders.

A national investment in developing, evolving, and maturing the software discipline will, therefore, benefit all information technology areas and most other application areas, as well as the related systems and products.

Findings:

- Basic and applied research in the software discipline should derive its motivation from real software problems, whether those problems are domain-specific or cross-domain.
- A field as young and dynamic as software still needs to put effort into establishing the principles and components of the discipline. For example, we have to ask such questions as: How do we obtain observable facts? What are the fundamental variables?
- Basic research problems involve issues of scalability, system evolution, and engineering process as well as developing tools and formalisms.

Recommendations:

- A significant research investment is required if we are to study real software problems. To increase our general understanding of software development, discover new principles that will help us build "no-surprise software," and validate theories of software construction, both new and long proposed, will require much time, effort, and funding support.

5. Mechanisms and Follow-Through.

Clearly, various government agencies will articulate high-level goals for research programs. However, it is important for the evolution of the discipline that these serve only as guidelines and motivations for research problems. Such guidelines should not be the final arbiter of what is good research. It is important that the researchers in the discipline should propose the research. Basic research proposals need to have the freedom to follow the best ideas and not focus on the short term. Quality control by the research community is important, and research proposals should be assessed for the quality of the ideas, how well they contribute to the evolution of the software discipline, how well they address important basic research problems, and how well they address the need for evaluating the work according to sound scientific principles.

Research review systems relying on peer review—the system used by the NSF, for example—generally manage well. They avoid wasting money on low-quality research. However, they may still miss some high-quality research because peer review can discourage risk, controversy, and preliminary ideas in favor of predictable progress. Thus, although peer review should be the basis for quality control, there should be some support for investing in risky, new ideas from researchers with excellent track records.

In administering investment in riskier programs, we may need to consider new mechanisms. An approach that bears investigation is the NIH "study section" model. In this system, a small group of senior researchers helps in making decisions for a large number of proposals, spanning a broad spectrum of areas. The system has the advantages of continuity (the same people do this work for several years) and the perspective and good judgment of the very best researchers. Study sections could be more likely than the one-shot small NSF panels to recognize high-quality, new, and risky ideas as worth investment..

We also believe that a wide variety of software research should be funded. By "variety" we mean over research areas (mathematical foundations, tool development, empirical studies, languages, operating systems, and human-computer interfaces, for example), as well as project styles (large and small, single and multiple-PI, and single and multiple-institutions). Research proposals should foster collaboration with other disciplines and with industry.

Since it is clear that software is pervasive, that it should adapt and adopt time-proven engineering techniques, that some of its major problems are project and business management, many research projects should be interdisciplinary, including not only computer scientists but also, as appropriate, engineers, and researchers from business schools. At the same time choices of research subjects should be discussed with reputable industry representatives who, contrary to often-heard prejudice, do care about the longer term prospects and health of their industry.

Of particular importance is the ability to facilitate and accelerate the transition of research across the chasm between software research and large-scale development in industry and government. There are many reasons for the problem, only some of which are technical. A major problem is the lack of incentives for academic software researchers and industry practitioners to collaborate and understand each other's objectives, constraints, and capabilities. There are many incentives for these groups to avoid each other. For example, academic research based on scaled-down computer science problems and simplifying assumptions—what Fred Brooks has called "tractable abstractions"— is much easier to generate and publish than research that tries to understand and address critical success factors for practitioners. On the industry side, applying yesterday's familiar solutions to today's problems is easier to do and defend than trying out risky new technology. Given this situation, creating incentives for understanding and collaboration between academic researchers and industry researchers and practitioners should be a high priority. However, care must be taken to avoid over-bureaucratic or artificial collaboration programs. Some examples of viable programs that could be scaled up or applied to software research are:

- Stimulating industrial collaboration in an expanded NSF Experimental Software Systems Program.

- The NSF Software Engineering Research Centers Program, which could have Centers oriented more towards software—with appropriate industry participation as a success criterion.
- The State of California's Micro program, which provides matching funds for industry-supported university research.
- Establishing and coordinating counterpart initiatives for experimentation with advanced research concepts and capabilities in Federal mission-oriented agencies (DoD, DoT, NASA, DoE, NIH, DoC, etc.) via HPCC-like mechanisms.

The major benefits derived from such initiatives would be:

- input from large projects to help guide the directions of the scientific research (keeping the science on the right track);
- avenues for technology transfer from the scientific research community to the development projects.

The software research projects discussed in this report require large grants sustained over many years. In particular, many experimental projects in software research involve several faculty members and numerous graduate students, in addition to postdocs, visitors, and staff programmers. We emphasize, however, that long-term grants require regular checkpoints for accountability. In fact, mechanisms for accountability can also serve as media for feedback and improvement.

Recommendations:

Funding agencies should sponsor software research programs that include:

- validation of concepts;
- indications of relevance;
- feedback mechanisms;

And meaningfully encourage

- academic/industrial collaboration;
- empirical investigation.

These research programs require large grants and should be sustained over several years.

Editor's Filler

Ready for some no-surprise software?

I'll take some – extra crispy please!

Obviously the original recipe isn't good enough!

Empirical Research in Software Engineering: A Workshop Marriott Hotel, Greenbelt, MD June 29-30, 1998 Sponsored by: The National Science Foundation Organized by: The University of Virginia The University of Maryland

Final Report Prepared by:
Susan S. Brilliant
John C. Knight

Executive Summary

This is a report of a workshop held in June 1998 to discuss the issue of empirical research in software engineering. The Universities of Virginia and Maryland organized the workshop, and it was funded by the National Science Foundation (NSF). The workshop was attended by representatives from academia, funding agencies, and industrial software development organizations.

Empirical research in software engineering is the observation of some aspect of software development in an experimental sense. The observation might be of an existing activity employing accepted techniques or of the application of new techniques. Clearly, this type of research is best performed on real development projects with professional developers although much of value has been learned by doing experiments with student developers. To ensure adequate input from professional developers, several representatives from industry attended the workshop who both understand research and have experience with development.

Empirical research is important to the software engineering field because the results of such research both help to characterize the technical problems with which the field is concerned and evaluate new techniques in a relevant context. In the view of the workshop organizers, insufficient empirical research in software engineering is being conducted despite the need and commendable efforts by funding agencies such as the NSF. The reason for holding this workshop was to review the situation and seek ways of enabling more empirical research.

The conclusions of the workshop are quite detailed. Many of the conclusions are suggestions to researchers of ways to make empirical research more successful. There are also ideas for industrial organizations and funding agencies. Beyond the detailed conclusions, several general and important observations came from the workshop discussions. Specifically:

1) There is a significant need for empirical research in software engineering. Many important issues faced by the community can only be addressed by experimentation.
2) Experimental work is complex and expensive to perform.
3) Although there are exceptions, in many cases industry does not perceive a significant benefit from working with academic researchers in joint activities of an experimental nature.