

Preliminary Report

*NSF Workshop on the Science of Design:
Software and Software-Intensive Systems*

Airlie Center, November 2-4, 2003

Edited by Kevin Sullivan
University of Virginia Department of Computer Science
February 10, 2003

I.	The Workshop.....	1
A.	The Need for a Science of Design.....	1
B.	Scope of the Workshop.....	2
C.	Workshop Objectives.....	2
D.	Invitational Process.....	2
E.	Emergent Outcome.....	2
II.	Findings.....	3
A.	Education.....	3
B.	New Composition and Decomposition Techniques.....	3
C.	Promoting Creative Exploration for Good Designs.....	4
D.	New Methods for Evaluating Complex Tradeoffs.....	5
E.	Design Languages and Analysis Tools.....	5
F.	Evolutionary Perspectives on Design.....	6
G.	Systems that Bend Under Stress.....	6
H.	Innovations in Type Systems.....	6
I.	Synthesis of and from Mathematical Specifications.....	7
J.	Tolerating Imperfect Information.....	7
K.	Integrated Problem Framing and Solving.....	8
L.	Contextualized System Design.....	8
M.	Design for Value.....	9
N.	Empowering End User Designers.....	9
O.	Archives of Software Design.....	10
P.	Empirical Studies of Designers.....	10
III.	A Note on Two Unresolved Tensions.....	11
IV.	Conclusion.....	11
V.	Bibliography.....	11

I. The Workshop

To date, no major research effort has addressed the science of design in a comprehensive, systematic manner, with a particular emphasis on software and software-intensive systems. In the context of a stated intention of the National Science Foundation's Directorate for Computer and Information Science and Engineering to establish a crosscutting research theme on the *Science of Design*, the editor of this report applied for and received a National Science Foundation grant, number CCR-0346938, to organize a workshop on the topic. The workshop, *Science of Design: Software and Software-Intensive Systems*, was held at Airlie Center, from November 2–4, 2003.

A. The Need for a Science of Design

Advances in software and software-intensive system design over the last fifty years have created deep knowledge and great benefits for our society. Nevertheless, long-standing technical challenges remain, and the march of information technology and other forces continue to widen the gap between software and software-intensive systems in practice and our underlying scientific and engineering knowledge and educational methods. We face hard and interesting problems. Left unsolved, they threaten us. Solving them would greatly improve our lives. Software has enabled miraculous applications—improving health, education, environment, our economy, our security. To harness complex information technologies of the future reliably and economically, however, requires significant advances in knowledge creation and education.

It's already possible, for example, for a single teenager to cause hundreds of millions of dollars worth of harm, perhaps even billions, by launching a malicious computer program into the Internet. Critical software systems are largely developed by composing third-party software components, but we lack the means to predict and certify their properties alone or in systems. The emergence of significant programming expertise around the world is creating a new need for U.S. employees to operate at a much higher design level than in the past. Business value is increasingly concentrated in software assets, but we lack the means to reason about that value and to design for value. To date, limited and fragmentary knowledge of software and software-intensive system design has sufficed. The future, in which all complex engineering systems will be software-intensive systems, demands a deeper and more unified *science of design*, with an emphasis on that elements which is now the most central but the least well understood: software.

The goals of such an effort include the following: To enable the design of software-intensive systems as dependable and fit to purpose as mechanical, civil, and other engineering systems typically are today; to ensure that the values and goals of the people impacted by software-intensive system are accounted for rigorously in their design and promoted effectively by their operation; to develop rich behaviors at complex interfaces between software-intensive systems and their human and physical environments; to enable automated tools based on sound science to replace costly and error-prone human activities in software design; to leverage knowledge of design from other fields; to develop design methods for emerging technology platforms; to understand how to design software-intensive systems as social agents; to understand the characteristics of successful designs and how they are achieved; to learn how to represent designs at a much higher-level than code but nevertheless rigorously embody the key constraints; to discover how multi-disciplinary communities of designers and users can communicate to jointly advance design processes; to devise methods to increase the design content of practitioner code and so increase the degree to which working with the code is working at the design level; and, crucially, to educate students in the fundamentals of design in all dimensions, with a particular emphasis on teaching effective software and software-intensive systems *design*.

B. Scope of the Workshop

The workshop organizers sought to define a scope satisfying several criteria. First, it had to be broad enough to consider the sciences of design, in general, in a sense such as, but not limited to, that advocated by Herbert Simon in his seminal book, *The Sciences of the Artificial* [1]. Second, the focus had to be on fundamental problems in software and software-intensive systems design, in particular. Third, recognizing that design theory is not a new idea and that it has been treated very seriously in disciplines beyond computer and information science and engineering, the workshop was designed to strongly encourage participation of design scholars from fields such as engineering design, architecture, and the social sciences, including economics.

It was also stipulated that the term *design* be taken in a design theoretic sense, referring to the whole process of problem framing, and conceiving, representing, evaluating, and evolving solutions; and not in the narrow sense sometimes used in software engineering, as referring merely to program structuring subsequent to specification and prior to coding. The broader, historically established interpretation recognizes the crucial importance of technical issues, but also the central relevance of extra-technical values, goals, conditions, constraints, and creativity.

C. Workshop Objectives

The workshop objectives were to develop positions on the proper formulations of a science of design, with a clear emphasis on software and software-intensive systems, addressing subject, scope, methods, norms, and the like; to identify major open problems and opportunities; and to develop guidance on research priorities for ten-year time frame.

D. Invitational Process

It was considered vitally important to open the workshop to participants from all relevant disciplines, both within and beyond computer science, and to have a broad range of viewpoints represented. A program committee of distinguished scholars was formed, with representatives from relevant areas of computer science and from engineering design and the social sciences. A call for position papers was disseminated widely, seeking contributions from both U.S. and international participants. Over 140 position papers were submitted. All papers were subject to review by all members of the organizing committee. Invitations were based on committee evaluations of the papers and on diversity considerations. The workshop had over 60 invited participants, in computer science, engineering design, architecture, and economics, coming from the United States, Europe, and Israel. NSF personnel and student volunteers were also present.

The program committee members were Carliss Baldwin, Harvard Business School; Fred Brooks, University of North Carolina, Chapel Hill; Clive Dym, Harvey Mudd College; Cordell Green, Kestrel Institute; Michael Jackson, Consultant; Alan Kay, HP Labs, Viewpoint Research Institute, and the University of California, Los Angeles; Gregor Kiczales, the University of British Columbia; Greg Morrisett, Harvard University; Jakob Nielsen, Nielsen Norman Group; David Notkin, University of Washington; Mary Shaw, Carnegie Mellon University; Kevin Sullivan, University of Virginia (*Workshop Chair*); and Richard Taylor, University of California Irvine.

E. Emergent Outcome

An organizing principle of the workshop was to avoid imposing a preconceived structure on the outcome, to allow the represented community to find its own voice. Therefore no attempt was made to impose tracks or categories on the discussion at the outset. Rather, the workshop process was intended first provoke the participants to identify key issues, and then to permit them to develop their positions in a form that could be incorporated into this report.

The workshop began in earnest the morning of November 3 with short talks from selected participants. The talks were expressly not keynotes, but were meant to provoke an initial round of discussions. Following the talks, participants were organized into parallel break-out groups, each given the same charge: formulate the problems and research ideas that will become the results of the workshop. Each group reported its findings in a plenary session. The reports were integrated by the workshop organizers to identify topics for an afternoon round of topic-specific break-out groups. Those break-out groups then reported back in an evening plenary session.

On the second day of the workshop, in a *marketplace of ideas*, individuals were asked to identify priority areas for research, consistent with the previous day's results, by holding up banner signs with keywords of their choice, and—reading each others' signs—to self-organize into affinity groups of up to about four members. Each group was then given several hours to produce a written document articulating the critical problems and research priorities in its area. Those short papers are the main source of the findings presented in the next section of this report.

Individuals were also asked to provide short written statements of their views, independent of their groups, and written dissents from group conclusions were also solicited.

II. Findings

This section of the report presents recommendations extracted from the work products of the workshop participants. There were sixteen areas in which participants formulated research problem envisioned research programs. A high priority was placed on the need to fundamentally improve education. The next areas ten are technical. The following three address issues concerning people. The last two see a need for empirical studies of designs and designers.

A. Education

The point on which workshop participants agreed most is that we don't teach software and software-intensive system design well. The most significant problem is that we still don't understand the field well enough. A second problem is that we don't teach very well what we do know—whether from within computer science and engineering or from fields such as engineering design and architecture. The participants placed a high priority on improving education, to train not just programmers and experts in component technologies but software and system designers. The National Science Foundation was seen as perhaps the only funding organization with a mandate to support this sort of development. A recommendation emerged that curriculum development and evaluation for software design be funded, addressing all levels of software development. It was argued that teachers should be able to state explicitly what the goal of each step in a design process is and what outcomes it will produce; that teaching should address questions of how to compare alternative designs at all levels; and that curricula should comprise interconnected courses related to research programs in software design. Since the evaluation of such efforts clearly requires a deep understanding of the subject matter, it was suggested that conventional educational evaluators are less appropriate in this context than peer evaluators.

B. New Composition and Decomposition Techniques

The structuring of problems and solutions was seen as an area in which past progress has been great, but in which, driven by the tremendous complexities of present and future systems, fundamental progress is required for the future. A recommendation emerged that research should be strongly encouraged on novel approaches, leading to advances in our ability to decompose and compose models and assumptions about problem worlds, requirements and specifications for software machines, and software components themselves. Participants saw a compelling need for major advances in our abilities to make appropriate decomposition and composition decisions.

Criteria for evaluating design structures are especially needed. We need for methods for evaluating tradeoffs between standard properties, such as function, performance, and cost; non-standard properties, such as flexibility and scalability; and system and context characteristics, such as complexity, architecture, uncertainty, and emergence. Research was recommended in at least three areas: non-hierarchical decomposition; structural mappings across representational domains; and in the development of new varieties of powerful architectural mathematics.

First, research is needed on structuring approaches that transcend hierarchical decomposition. Such techniques include those that map descriptions between representational layers, and methods that enable crosscutting composition and decomposition.

Second, research is needed on techniques to relate descriptions across layers, such as specification and code. The problem is to be able to reason and ask questions about low-level code in terms of higher-level constructs. Approaches to code querying might include meta-programming notations for representing mappings; abstract representations and to-code mapping techniques; static and dynamic techniques for generating representations from code; and query languages that translate abstract queries into queries against databases of facts about source code.

Third, research funding agencies should *find and fund people* who demonstrate the potential to develop new varieties of dynamic architectural mathematics. The creation of powerful-but-tiny models of *math in the domain* as exemplified by Lisp and Sketchpad provided significant advances in past decades, and possible models for future systems. Lisp, for example, linked with mathematical models (it was one), served as its own meta-language, and enabled tiny and powerful universal computing. Similarly, Sketchpad had three great inventions (realizations of ideas): interactive constructive graphics; dynamic modeling, simulation, constraints and solving; and object-oriented software structuring. Sketchpad chose not to try to solve constraints symbolically but instead to bring errors within acceptable tolerances, which could be done by iterative numerical methods. This approach led to dynamic system stability of late-bound loosely coupled and slightly quivering integrity rather than a static formally deduced one. LISP and Sketchpad were the first startling examples of dynamic architectural mathematics. It's math, but a new kind of math, where a critical property is that it is about, and represented in, a dynamic system that can do feedback processes on itself. These math ideas are quite a bit more analogous to various way of thinking about mathematical *analysis* than mathematical *logic*.

In physical architecture, one can build large structures by simply piling up material (like the large pyramids in Egypt), or one can build similarly sized structures with a fraction of the material by inventing and building (ecture-ing) arches (literally architecture). In matters of complexity, architecture dominates material. Most software today resembles the pyramids, but there do exist extremely important instances of the latter, both in the past and today. Research funding agencies *should find and fund these ideas, and set up new processes to create centers of growth for the deeper ideas of this kind of design.*

C. Promoting Creative Exploration for Good Designs

A critical need is *enabling the effective exploration of many alternative and novel approaches to demanding design challenges.* Current technologies often drive designers to premature formalization and encumber them with costs that inhibit backing out of one alternative to explore others. We are plagued by premature codification of methods; tools that are method driven, not problem driven; and generic representations that incur unnecessary costs, force premature commitments, and conflate the critical with the incidental.

Part of the problem is cultural. An undue preference for being science can deprecate the creative, uncertain, unproven. Moreover, design is now interdisciplinary, typically meaning that it is done by remote and loosely allied teams. Complexity further compounds the problem.

Exploration of alternatives is easier and cheaper when complexity is low. The means of the past are unlikely to scale to the problems of the future; and they haven't.

Research is needed on new design methods supported by novel tools and admitting ambiguity, inconsistency, and imprecision. Four issues should be emphasized. First, *problem framing must be integrated with problem solving*. The problem is seldom a given or immutable. Indeed co-development of problem statements and solutions is the norm. Second, *creative solutions* must be fostered. This can be done by supporting knowledge of prior art, allowing analysis of failures and mining of prior developments for creative advances. Techniques such as remapping problems to other domains should be taught. Third, *support for sketching, visualizing, representing, and evaluating design ideas and concepts* should be provided. Support for exploratory and experimental prototyping is consistent with this emphasis. Critically, such formation techniques must be accompanied by techniques providing low-cost/low-commitment evaluation. The designer should not be compelled to commit to lots of details simply to enable evaluation. Rather, it should be feasible to conduct evaluations from low-fidelity representations. Fourth, we must provide *support for collaborative design*. Team based design has the promise of tapping the collective creativity. Such techniques must provide ways of supporting multiple points of views, combinations and associations of ideas, and encouraging associations across viewpoints.

D. New Methods for Evaluating Complex Tradeoffs

With requirements and constraints in many dimensions (product scope, quality, cost, time-to-market, packaging, globalization, standards compliance, usability, evolvability, etc.) it is hard to make appropriate design decisions and tradeoffs, particularly since designers tend to make decisions focused on their own areas of responsibility.

Much of today's research and education in software design is based on optimizing decisions in a single area, such as database design. While it is essential for software professionals to have specialized knowledge, it is also important to have perspective on how design decisions in a single domain interact with the myriad other decisions that go into product development. In today's commercial environments, the product manager in the marketing organization frequently lacks the technical knowledge needed to design and build the product, while the engineering manager often fails to fully understand the business issues that impose additional constraints on software development.

These issues have implications for creating a science of design focused on software. Researchers should investigate not only specific fields of design, but also how to address tradeoffs across fields, the effect of global constraints, and the influence of external issues, e.g., cost and time to market, on design processes.

E. Design Languages and Analysis Tools

Current design and modeling languages lack important features: expression of global requirements for privacy, quality of service, real-time constraints, and communication; expression of crosscutting properties that affect multiple views; capture of operational assumptions and user requirements; support for automated analysis of models against implementations; support for behavioral and rich interfaces for modules and components; tracking connections with code, particularly as implementations evolve; capture of design rationale; and support for comparing designs according to well-defined metrics.

Research should be encouraged to develop new languages and tool supporting such features. Continued work is needed in several areas including the following: analysis techniques such as model checking, theorem proving, abstraction, and static analysis; extracting models from

code using automated abstraction to enable the application of high-level analysis tools to source code; analysis techniques for specific domains such as security protocols, real-time and hybrid systems; a logical basis for formalizing domain knowledge and model integration with tool support; domain specific APIs; and libraries for design integration.

F. Evolutionary Perspectives on Design

Software systems differ from many other engineered systems in the necessity for rapid change to meet evolving needs. At the same time, anticipation of change in software-intensive systems is more difficult than in most other kinds of systems, in part because software is often used in unanticipated ways, revealing new requirements. In addition, the complexity of problems and solutions creates enough uncertainty that evolutionary approaches are often the most effective. Research is needed on evolution in software and software-intensive system design. We need a better, empirically-based understanding of phenomena of requirements change for software. Even incremental improvements can lay foundations for better design. We need better accounts of how design structures and evolution relate. Concepts from complex adaptive systems might be useful. We need advances to enable dependability properties to be maintained and re-verified at reasonable cost as systems evolve over time. We also need methods based on sophisticated co-evolutionary models of design, rather than simplistic waterfall-based models.

G. Systems that Bend Under Stress

Research should be encouraged on design approaches that acknowledge the undesirability of attempting to completely understand the potential errors, failures, and operating conditions of complex software systems. These new approaches would focus on augmenting software systems with redundancy, introspective capabilities, and repair or reconfiguration actions that enable the system to detect and recover from damage and adapt to changing operating conditions. An important aspect is developing designs whose structure minimizes failure and error propagations. Such research might involve program transformation techniques that help systems to ignore certain classes of errors and bound the range of possible behaviors. It is recognized that efforts in this dimension are inconsistent with traditional approaches to critical software systems dependability, which generally demand comprehensive hazard analyses and cases showing that all potentially critical failure modes have been addressed adequately. Research should address the conditions under which the respective approaches are most appropriate.

H. Innovations in Type Systems

It is impossible to decouple design from implementation. Design guides implementation, and implementation validates and informs design. To facilitate this interplay between these two activities it is important to narrow the gap between design and implementation by providing specification formalisms that express design constraints and providing tools that check conformance of implementations with these constraints.

Type systems have proved to be a very successful method for improving software quality. Types provide the means of formulating an abstract theory of the problem domain to gain a better understanding of its structure. Types in interfaces support modularity and re-use by constraining the contexts in which a component may be used, minimizing unintended interactions that lead to integration problems, and providing assurances about the properties of composite systems. Types are easy to write down, and easy to read, and in most cases can be mechanically checked to minimize the divergence between design and implementation by ensuring that the code satisfies its type specifications. Types are part of the code itself, evolving with it, and checked whenever the system changes.

Ongoing and future research on type systems is concerned with extending their expressive power to capture richer classes of system properties without sacrificing their advantages relative to full-scale verification systems. A good type system should mediate the interactive dialog between design and implementation. The programmer engages in a dialog with the type checker, modifying the implementation to satisfy the constraints imposed by the checker, or revising the specifications to better reflect design intentions. A good type system should support compositional development, allowing a component to be specified and checked once, and re-used in many contexts.

Research is needed on the development and use of type systems to narrow the gap between abstract “designs” and concrete implementations. Types in this context are viewed as specification formalisms that express design constraints, with tools to check conformance of implementations. Future work might address the following issues: extending the power of types to express not only structural but behavioral constraints; extending types to constrain information flows in programs; types as constraints on relationships among components; the development of domain- and application-specific type systems, including validation of custom type systems, the composition of type systems and of components adhering to different type systems; and types to constrain and check end-to-end system properties.

I. Synthesis of and from Mathematical Specifications

Research should be encouraged into methods to factor design knowledge into forms that can be manipulated by computers to enable automated generation of code and other program representations including proofs of program properties from high-level models. Doing so could lead to significant improvements in the management of domain-specific software complexity and the reduction of such complexity. Basic science is needed to create general paradigms and theories for program synthesis, libraries of representations of design knowledge, and tools that ground these paradigms in real-world applications. Demonstrations of the scalability of the paradigms, theories, and tools are essential.

The semantic integration of heterogeneous models presents challenging and interesting problems. Leveraging formal specifications in late development stages, including testing, is ripe for development.

J. Tolerating Imperfect Information

Research should be encouraged to improve our understanding of how to make rational design decisions in the face of imperfect knowledge. Imperfections include missing information, statistically uncertain information, and inconsistent information. Imperfections in knowledge of both environments and designs are unavoidable. Source code does not solve the problem. Progress almost never depends on perfect knowledge. Rather, improvements in knowledge generally confer benefits, but such knowledge often comes at a cost (e.g., the cost of testing). Design methods and mechanisms are needed that explain, account for, and promote effective design in the face of imperfect knowledge and to reduce the cost of acquiring knowledge. Questions include when to invest in information (e.g., by testing, prototyping); approaches to reasoning in the face of inconsistency; stochastically optimal decisions under uncertainty; theories that allow one to rely on aggregate reasoning about overall behavior rather than exact reasoning about details; empirically verifiable statistical regularities in large-scale design activities and structures; and theories that help explain how much information is enough to make risk-acceptable, informed, and cost-effective decisions.

K. Integrated Problem Framing and Solving

Design isn't restricted to any specific discipline, such as art or architecture, but is a broad human activity that pursues the question of "how things ought to be," as compared to the natural sciences, which study "how things are." Designers solve problems. But apart from problems in school, most *problems in real life are encountered, not given*. For these problems, understanding the problem *is* the problem. Real-life problems must be *framed*, a process in which the important objects are determined and desired outcomes are defined.

Today designers of software and software-intensive systems too often go awry for lack of a proper understanding of the problem to be solved—of the requirements for a system. Research should be encouraged to foster the development, evaluation, and promulgation of new design approaches in which problem framing is integrated with problem solving; creative solutions are encouraged; sketching, representing, visualizing, and evaluating design concepts is supported, along with collaborative design activities.

A critical need is *enabling the effective exploration of many alternative and novel approaches to demanding design challenges*. In so doing we must achieve the ability to manage the tension between constraint and freedom, between rigor and relevance. Our current technologies drive designers to premature formalization; they encumber designers with costs that inhibit backing out from one alternative and exploring others. We are plagued with premature codification of methods, tools which are method driven, not problem driven; generic representations which incur cost, force premature commitment, and conflate the critical with the incidental.

L. Contextualized System Design

Most real systems are *lumpy*: coexisting sets of hard and soft subsystems and interactions, where *hard* subsystems include software and physical objects and *soft* subsystems include fluid, idiosyncratic but non-random entities such as individual people, organizational groups, policies and practices. Relatively speaking, we know how to represent, reason about, and explore designs for the hard components far better than we know how to represent, reason about, and explore design for soft components.

Current systems are built on an infrastructure designed to support a very different set of problems than we now confront. Current approaches have grown out of a concern with issues such as display management for individual applications like word processing. Our computing environments are still as file-based as they were in the 1970s. In contrast, applications over the next ten to fifteen years will involve human and organizational concepts such as trust, commitments, ownership, responsibility, negotiation and delegation. We will need to describe systems at all levels of detail in terms of these concepts. For ubiquitous, mobile, wearable, and distributed personal computing systems, we need runtime concepts and interaction primitives that mesh better with human activities. In addition, systems are and will be used increasingly to support exploratory, open-ended and ongoing problem solving in addition to discrete, rule-driven problems.

Research should be encouraged on integrative approaches to representing and exploring properties of hard and soft facets of a system in which *context* is addressed at the level of primitives. *Context* is often seen in software engineering as comprising a set of external constraints surrounding a system. The etymology of the word is revealing, however: Context is *woven through* a system. A science of design that gives context central significance needs to go beyond well-known techniques for representing context such as thick descriptions, storyboards, etc. toward those that can integrate with more formal abstractions.

There have already been several attempts to develop rigorous description languages on a foundation of social or organizational primitives. Some legal expert systems research has used deontic logic as a foundation. Multi-agent systems have been constructed using speech-act protocols with formal definitions of such concepts as “promise”, “request”, “agree”, etc. The security community has detailed (though narrow) models of trust and trustworthiness, authority, authentication, etc.

One promising approach is scenario-based design. Social and organizational primitives could be used in conjunction with scenario-based design so that both soft and hard subsystems of a complex software-intensive system as well as their interactions could be modeled as conversations or negotiations rather than mere information processing. As a complementary strategy, these primitives could even be built into future operating systems so that they were visible during execution.

Scenarios exhibit several shortcomings as often used:

- A scenario has meaning only to the extent that its constituent steps do. By proposing scenario-based design in conjunction with social primitives, we are providing such a vocabulary.
- Scenarios are linear, whereas much activity is interestingly concurrent and interleaved with other contexts.
- Scenarios are incomplete examples of activity and may not cover the critical activities.

Modeling context in a set of new socially informed abstractions is principled and predictable. However, it is necessarily self limiting when compared to an open-ended approach such as design ethnography which may discover unanticipated activities and relationships.

Reifying important contextual concepts will not guarantee that resulting designs will be convivial. It is not clear how people will understand or enact such activities as delegation or negotiation with computational agents. Early experience in CSCW (e.g. The Coordinator) showed that when made explicit, implicit activities can impose a burden on users.

M. Design for Value

Research should be encouraged to develop and evaluate concepts, theories, models, and technologies to incorporate quantitative consideration of uncertain costs, benefits, risks, and opportunities in software design. Issues include design for value (e.g., through lock-in, design for change); technical decision-making informed by value considerations (e.g., how much to invest in architecture); the cost of coordination in design; cost of ownership; cost of gathering information to support analysis; costs and benefits of analysis; risks and opportunities associated with imperfect information; and multi-dimensional measures of cost, benefit, risk, and opportunity.

N. Empowering End User Designers

A growing trend is the delivery of programming capabilities to end-users—in spreadsheet languages, scripting languages embedded in mass-market tools, rich configuration capabilities, integration mechanisms, and so forth. Software design knowledge today is not geared to helping end users to design the software that they need. Research should be encouraged on theories, methods, and technologies aimed at empowering these millions of end user programmers.

The number of end-user programmers in the U.S. is expected to reach 55 million by 2005, as compared to only 2.75 million professional programmers [2]. Much of the software produced by these 55 million people will be *everyday software*. Their needs and skills will be very different from those of the current generation of professional software developers. Many of

their needs for computing capability will be personal and transient; it will often be cost-effective for them to notice failure and recover rather than paying the cost of high dependability; they will be more conversant with computers and software than the current population, but they will not be trained with specific software development skills.

Current design methods for software-intensive systems will not suffice for the needs of everyday users for the next decade. In order to think sensibly about this change, it is imperative to consider the incentives and motivations of the 55 million end user programmers. For them, software is instrumental, not an end. They must find it personally cost-effective to create and manage their own software solutions. Current programming objectives such as complete correctness, full specification, and high dependability will not serve them well, because the value of those properties often does not exceed the personal and financial costs of establishing them.

Economic benefits and costs that have not traditionally been salient to software engineering decisions are of first-order importance to end-user programmers. These benefits and costs include such things as learnability, cost of owning, transacting, and maintaining, fitness-to-purpose, variety, even *coolness*. They are different from the classic requirements and constraints that are the focus of much software engineering effort today. But these value drivers of every-day software will be of first-order importance to the army of end-user programmers that is emerging and thus are an appropriate focus of research and research funding today.

Research should thus be encouraged to development of a theory and practice of software system design that incorporates economics as a first-class driver of decision-making; guides and explains technical decisions for software that is *good enough*; and supports the development of software systems that will create value for and convey value to end-user programmers.

O. Archives of Software Design

Today there is an insufficient basis for accumulating knowledge about good designs, designers, and design process. Research and development should be encouraged leading to the creation of durable archives of exemplary software designs, including paradigmatic successes and failures, the artifacts themselves, retrospectives from designers, and information on the design processes leading to the artifacts, such as documents and e-mail. Such archives should be designed to support empirically-based analytical and critical studies of artifacts and other data, with the common repository supporting analyses of different kinds by different disciplines (e.g., a given artifact might be interesting for its high reliability but uninteresting for its usability). The chief criterion for the inclusion of candidate artifacts in the archive would be that a system have been delivered and used by people other than who developed it.

P. Empirical Studies of Designers

Research is needed to extend our empirical knowledge of the activities of designers in practice. What do software designers (architects, developers, etc.) actually *do* and what breakdowns in what they do lead to the problems with products (schedule, cost, reliability, usefulness or usability, etc.)? Different people have many different opinions about where the breakdowns are (and therefore where the solutions lie), but decades of research in HCI have confirmed that “the user it not like me”, and that retrospective memory of process followed is also unreliable (i.e., “I am not like me”). Intuition is inadequate to provide valid answers about where current problems arise. Well-structured observational methods should be used to build sound foundations for understanding both the processes involved and the bases for *good* design decisions, together with approaches to synthesizing the resulting data. Design cognition studies could be valuable in relating exemplary design to its mental, organizational, and cultural enablers. Improved methods of disseminating design knowledge to practicing designers are badly needed.

Researchers should investigate software development, in situ, using well-established field research techniques by technically competent, interdisciplinary teams that include behavioral scientists (psychologists, social psychologists, ethnographers, etc.) as well as computer scientists. A condition that should be imposed on this research would be that the raw data (e.g., interview transcripts, artifacts, etc.), as well as the distilled reports and recommendations, must be made available to the community so different groups can mine those data for evidence of breakdowns in different areas. This will require effort to make the raw data anonymous, which must be budgeted in proposals. Reviewers should include experts in field work, because it is easy to propose naïve or mediocre projects. These must be weeded out by empirically-sophisticated reviewers.

III. A Note on Two Unresolved Tensions

Concerning the question, *Can there be a science of design for software and software-intensive systems, and, if so, what should it be*, there were differences in two dimensions. First, several attendees argued that there can be no science of design in Simon’s sense, because science is concerned with discovering properties of the given world while design is concerned with making new things. The point was raised but there was no attempt to debate it. The issue was resolved by a tacit agreement that *science of design* could be interpreted to mean an organized body of rigorous knowledge about how to make things (software and software-intensive systems).

Second, and more seriously, there were differences on the proper scope of funded research activities. The diversity of backgrounds at the workshop ensured that the group as a whole would take a broad perspective, addressing not only technical issues but cognition, economics, social and organizational parameters. Nevertheless, some participants argued that our ability merely to develop technically sound software—especially innovative software—is still so dangerously poor, that by far the most pressing need, and the one most appropriately addressed by researchers, is to make progress solving that foundational, fundamentally technical problem.

IV. Conclusion

Design is the key to harnessing the exploding power of information technology, but we still lack a science of design, in general, or for software-intensive systems, in particular. workshop results described herein are perhaps a step toward the formulation of a research initiative to establish, disseminate, and exploit such a new science of design.

V. Bibliography

- [1] H. A. Simon, *The Sciences of the Artificial*, MIT Press, Cambridge, MA, 1969.
- [2] B. Boehm et al., *Software Cost Estimation with COCOMO II*, Prentice Hall, 2000.