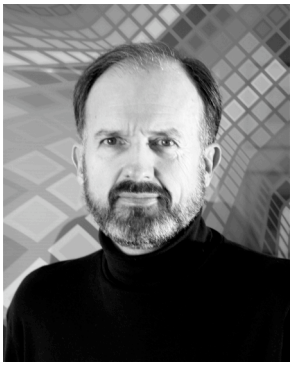
Software Design and Architecture

The once and future focus of software engineering

Richard N. Taylor and André van der Hoek



Richard N. Taylor is a Professor of Information and Computer Sciences at the University of California, Irvine. He received the Ph.D. degree in Computer Science in 1980. His research interests are centered on software architectures, especially event-based and peer-to-peer systems and the way they scale across organizational boundaries and decentralized applications. Professor Taylor is the Director of the Institute for Software Research, has served as chairman of SIGSOFT, chairman of the ICSE Steering Committee, and was general chair of FSE 2004. Taylor was a 1985 recipient of a Presidential Young Investigator Award. In 1998 he was recognized as an ACM Fellow and in 2005 was awarded the ACM SIGSOFT Distinguished Service Award.

André van der Hoek is an associate professor in the Department of Informatics at the University of California, Irvine. He holds a joint B.S. and M.S. degree in Business-Oriented Computer Science from the Erasmus University Rotterdam, the Netherlands, and a Ph.D. degree in Computer Science from the University of Colorado at Boulder. His research focuses on understanding and advancing the role of design, coordination, and education in software engineering. André is the principal designer of the new B.S. in Informatics at UC Irvine and was honored, in 2005, as UC Irvine Professor of the Year.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

Software Design and Architecture

The once and future focus of software engineering

Richard N. Taylor
Institute for Software Research
University of California, Irvine
Irvine, California 92697-3455
taylor@ics.uci.edu

André van der Hoek
Institute for Software Research
University of California, Irvine
Irvine, California 92697-3455
andre@ics.uci.edu

Abstract

The design of software has been a focus of software engineering research since the field's beginning. This paper explores key aspects of this research focus and shows why design will remain a principal focus. The intrinsic elements of software design, both process and product, are discussed: concept formation, use of experience, and means for representation, reasoning, and directing the design activity. Design is presented as being an activity engaged by a wide range of stakeholders, acting throughout most of a system's lifecycle, making a set of key choices which constitute the application's architecture. Directions for design research are outlined, including: (a) drawing lessons, inspiration, and techniques from design fields outside of computer science, (b) emphasizing the design of application "character" (functionality and style) as well as the application's structure, and (c) expanding the notion of software to encompass the design of additional kinds of intangible complex artifacts.

1. Introduction

Design is the central focus of software engineering. Design is both a verb and a noun. It is a key thing we do and that we produce.

Such crisp statements will alternatively strike one as obvious or, perhaps, as parochial – if not incorrect. Yet if we consider what software engineering is, namely a practice directed at the production of software systems, then design is seen at its heart, as it is in any other productive enterprise, whether the creation of skyscrapers, automobiles, toasters, or urban regions.

Not surprisingly, then, many software engineering researchers, or those acquainted with software development, have studied and written about software design over the past forty years and more. Fred Brooks included in his 1975 list of "promising attacks on the conceptual essence" the growing of great designers [21]. Peter Freeman, in 1976 [31], said "Design is relevant to all software engineering activities and is the central integrating activity that ties the others together."

Design will *remain* the focus of software engineering. Herb Simon, in his classic, *The Sciences of the Artificial* [75], includes a discussion of design in the context of "artificial" fields, such as software development, saying:

> "The artificial world is centered precisely on this interface between the inner [the means] and outer [the task] environments; it is concerned with attaining goals by adapting the former to the latter. The proper study of those who are concerned with the artificial is the way in which that adaptation of means to environments is brought about – and central to that is the process of design itself."

Put in software engineering parlance, the outer environment is the world of requirements, goals, and wants; the inner environment is the set of software languages, components, and tools we have for building systems. As software engineering researchers, we are always "raising the floor" – creating new levels of infrastructure upon which new developments may be built. In Simon's terms, the "inner environment" or the "means" is ever changing and expanding. As the floor rises, however, so do our desires and aspirations. Though achievements in improving design have been obtained over the previous decades, new challenges for design will thus continuously arise.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

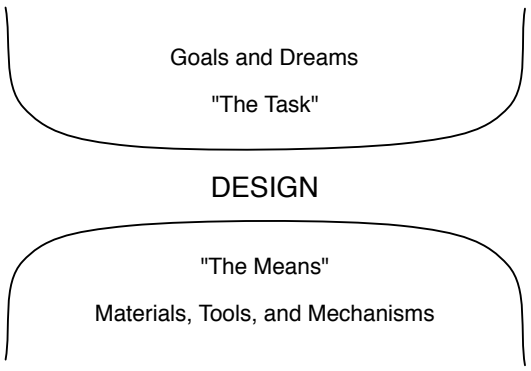


Figure 1. The continuing place of design.

Nonetheless, at a suitably abstract level the *challenges* for software design today are the same as they were forty years ago. They are the intrinsic challenges of design: How to create artifacts to obtain goals, how to represent new conceptions, and how to analyze them. Brooks made this observation twenty years after the original Mythical Man-Month was published. He said the distinctive concerns of software engineering now are exactly those he set out earlier, namely "How to design and build a set of programs into a system; How to design and build a program or a system into a robust, tested, documented, supported product; How to maintain intellectual control over complexity in large doses." [20] (pg. 288). Ten years after *that* statement it is still true.

Arguably all the major threads of software engineering research are directed at improving our ability to meet the challenges of designing software. Work in requirements engineering contributes to Simon's "outer environment"; process research addresses the coordination of all activities focused on creating, implementing, and evolving designs; empirical studies improve our ability to assess design artifacts and the processes by which they were created; analysis research improves our ability to assess candidate designs; work at the patterns and frameworks level improves our ability to realize designs in source code; and so on.

Though a focus on design has been, and will be, the central issue in software engineering, the type of design on which our energies have been focused has been rather lopsided. Our focus has largely been directed at the design of software *qua* software. That is, we focus on the structure of software and its attributes, such as considering what components and connectors comprise a system, and what constraints govern their interactions. A lesser role in software engineering has been assigned to the design of software as it exhibits characteristics to its users. For instance, what "interactive feel" does the application give to its users? What "style" does it exhibit? What is its branding, or distinctive behavioral character? Using the analogy of automotive design can make the distinction clear: design research of the first type is directed at the mechanical organization and structure of the vehicle; design research of the second type is directed at shaping the car's appearance, performance, sound, and smell. Doing a good job of one type of design does not imply doing a good job with the other, yet they are intrinsically interrelated. Both are important, and are legitimately the subject of (software) design research.

Work on design of the first type has certainly yielded a wide range of important results over the past several decades. Numerous development methods have been espoused, many based upon the articulation and application of design "principles" such as modularity and planning for change. Means for representing designs have been devised; domain-specific approaches have been created and supporting tools supplied. In recent years, particular advances have been made with regard to product families and the careful specification of system architectures.

Work on design of the second type has often been ignored by software engineering researchers, and instead relegated to either other sub-disciplines of computer science, especially human-computer interaction researchers, or simply left to practicing engineers in industry.

Design of both types is increasingly recognized as a critical corporate and national asset. Designing of products is seen as an activity that cannot be effectively off-shored – regardless of the shore on which one is standing. Design can be a differentiator that determines an organization's success. The ability to design effectively is typically the partner of the ability to innovate.

The remainder of this paper seeks to explore how and where to advance from the state-of-the art. We proceed by first examining some major historical threads of design research, then highlighting a few notable current trends. These sections are not merely background; by straightforward implication from what we describe, some key directions for software research emerge. Drawing from the perspectives of these two sections, we then examine the character of design in more detail. The remainder of the paper explicitly lays out a set of further research directions, and concludes with some challenges and a "long view" of the promising future of software design.

2. Paradigms and Persuasions

The past forty plus years of design research can be taxonomized many ways, such as by describing the history of models, methods, and tools over time. These topics are not independent, however. Rather, the background or disciplinary orientation of an individual or group tends to bring along a set of beliefs, choices, and approaches. Any such perspective, then, drives specific choices in a variety of areas.

Prescriptive Design Methods

Perhaps most familiar to the software engineering researcher is the perspective of the "software methodologist". Typified best, perhaps, by work in the 1970's and 80's by such authors as Yourdon [81, 82], Jackson [41, 43], and Parnas [61-65], this strand of work focuses on the first type of design discussed above, the design of software as the artifact of interest, and is an approach that states a principle, then prescribes how to design based on that principle.

Much of this work has been top-down in style; principles of design are articulated ("separate abstractions", "use information hiding", "refine higher-level abstractions into a set of lower-level abstractions", etc.) and then either methods for employing those principles or notations which highlight or support application of the principles are developed.

One influential strand of work began with Russ Abbott [8], and was then expanded upon and advocated by authors such as Grady Booch [17]. This design approach is, roughly, "design by simulation", in which the software application is straightforwardly designed by creating software objects that correspond to entities in the real world, and whose methods correspond to actions in the real world. The work contributed to object-oriented design, and became in a broader context to be supported by methods such as the Rational Unified Process [51], with designs represented in the Unified Modeling Language (UML) [52].

Notations

Notations have been a part of software design since the beginning. Any time design thought is externalized, such thought must be written down in some structure or form that supports interpretation at a later time by others, oneself, or a computerized program. It is no surprise, then, that notations continue to serve as a primary driver of research in the community.

Notations range from informal conventions that are established on-the-fly by a group of designers engaged in a design exercise to precise formalisms that are standards for the field. Two primary concerns in the formulation of new notations are expressiveness and usability. Expressiveness concerns what aspects of a design can be captured in the notation; usability concerns the fluidity with which designers can work with the notation. Though both factors are equally important, the primary driving force behind the development of most new notations has been expressiveness – adding modeling capabilities, often for a particular analysis purpose.

Extensibility is a required property of any modern notation, as it is now common knowledge that no standard notation can fulfill all modeling needs. UML profiles are perhaps best known in this regard, with a host of profiles publicly available that address a broad variety of modeling concerns.

The Wisdom of Experience

Still focusing on the design of software, but coming at the problem from essentially a bottom-up perspective, is a strand of work focused on capturing the lessons of experience in such a way that future designs can be guided. The work on "design patterns" is typical of this strand. While the "Gang of Four" patterns [33] are directed at the programming language level, the concept can be applied at any level of abstraction, including requirements (where the experience may be captured as "frames" [42]), whole-concept system structure (where the experience is captured as domain-specific software architectures [10, 38, 79]), and generally at the level of system components and connectors (where the experience is captured as styles and architectural templates [9, 34]).

Methods for analysis and restructuring of software may reflect insights from this research strand, such as are found in refactoring analysis and reverse engineering.

Knowledge representation and design rationale research may contribute to the effective employment of design based upon the wisdom of experience.

HCI Design

Innovation in product design and distinctiveness in product design have long been argued as contributing to product (and corporate) success [45] – though such innovation is no guarantee of commercial success. The software engineering literature is relatively silent on the matter of such design, possibly because researchers view the subject as too domain-dependent, and hence exclusively the focus of developers within that domain. Even if one assumes that, it is surprising that software researchers have not focused on the methods by which innovative domain-specific designs are created.

The exception is user interface design. Often relegated to (at best) the fringes of software engineering research, human-computer interaction research includes a focus on techniques for developing user interfaces which effectively engage and satisfy the user. Some lessons of such work are captured, for instance, in patterns for web site design, as found in the work of Landay and colleagues [80]. This work is partially bottom-up, in the manner described above, but also reflects cognitive understandings of how people interact with web sites and undertake electronic commerce transactions.

The importance of considering this field is indicated, in no small measure, by the repeated failure, over many years, of standard software engineering top-

down design techniques to create pleasing and effective user interfaces.

Design Outside of Software

The field of design, and design research, outside of software engineering and computer science is enormous. While both types of design discussed above are within this wider world's view, a greater emphasis is found on the second type – user experience. Less work is found on representational issues; since physical objects are being designed the representation means (typically sketching) is natural. More work has been directed at methodological approaches, with [44] a classic example from the domain of industrial engineering. The software patterns work, however, derives its name, at least, from work in building architecture [11].

The relevance of lessons from design "out there" to software design has been noted by many. Participatory design has been advocated by some in the HCI community [35], and, arguably, software engineering's "agile design" draws from some of its key elements . Less directly, the capabilities of CAD systems like CATIA have been an inspiration throughout.

A further perspective on design exhibited by the larger design world, but which software engineering has mostly ignored (or scorned) is that of design as art. While Donald Knuth unabashedly titled his monumental series, "The Art of Computer Programming" (e.g., [46]) and promoted "literate programming" [47], we have not developed a practice of critiquing the aesthetics of any kind of software design, of endeavoring to instill an appreciation for elegant software design, or of providing public forums for the promotion and recognition of excellence in design[1].

Cognitive and Social Strategies

A final strand of design research is exemplified by the work of Donald Schön [72], and found in software engineering in the work of, for example, Fischer. A key perspective emerging from this work is what Schön terms reflection: the designer reflects upon the process and upon the product. Design in this view emerges as a "conversation" between the materials (the external constraints on the design, the initial sketches attempting to develop a solution) and the designer.

Design, however, is not a solitary activity. Teams of designers interact, varied stakeholders participate, and broader communities of practice [29] exist within which individual designers perform their work. Design, thus, is a social process, one of information exchange, learning, creative and cognitive stimulation, and conversation – all aspects that must be taken into account.

[1] The ACM Software Systems Award is an exception, though it is not widely promoted – certainly not in the software engineering community.

3. Contemporary Currents

Contemporary currents in design sometimes add important perspectives to the schools of thought described above, as well as combine, apply, and refine them.

Agile Methods

While for some "agile methods" are a step backwards in the history of software design research – because the design exists only in the code –, the agile community has emphasized three important design practices. First, it is an example of the application of participatory design: by involving the user throughout the iterative development process, the product is continuously shaped to meet the user's needs. Secondly, test-driven development makes the design of functionality of equal importance to the design of system structure, which represents a rudimentary integration of the two types of design we discussed in the Introduction.

Lastly, implicit in the agile approach is that the process of design continues throughout development. With little extrapolation, design can be seen to continue throughout the life of the product – a characteristic not shared with many products from the realm of physical product design.

Aspect-Oriented Software Design

The original aspect-oriented programming paper states "A design process and a programming language work well together when the programming language provides abstraction and composition mechanisms that cleanly support the kinds of units the design process breaks the system into." It then makes the argument that programming languages must conceptually distinguish components (units of a system's functional decomposition) from aspects (system properties that cannot be cleanly separated and instead crosscut components) [54]. This view of AOP strikes an important chord with design, as separation of concerns is one of the leading approaches to tackling a design problem's complexity. Unfortunately, this design-oriented perspective seems to have given way to AOP language minutiae and a focus on "aspectizing" any and all software artifact. Nonetheless, the critical role of the programming language in the design process persists, and AOP has and continues to have an impact as such.

Design Analysis

One of the reasons we design is to reduce risk by enabling prediction of system properties. Not surpris-

ingly, the field has devised numerous ways of performing design-based analyses. Many of these have been described, for example, in the SAVCBS workshop series (Specification and Verification of Component-Based Systems). Most of these analyses concern externally visible properties, such as reliability, real-time constraints, or concurrency, although a fair amount of work also concentrates on properties internal to a design, such as structural quality or reusability [71].

Different design representations are suited to different kinds of analyses; new analyses may require new notations to be used.

Component-Based Design

A desire to structure large-scale business applications in terms of standard, reusable components continues to drive a non-trivial part of the industrial software landscape. While each guarantees somewhat different properties as emphasized by somewhat different usage scenarios, component-based design, model-driven architecture, and web services all can be grouped as addressing this desire in a similar manner. In fact, they can be seen as evolving from one another, with component-based design focusing on reuse of the individual component, model-driven architecture on standardization of components into reusable middleware [30, 40], and web services on reuse of components and middleware across distributed and decentralized applications.

Of course, different components do not magically fit together. A particular challenge is to design the "glue" that bridges mismatches in functionality, interfaces, and interaction paradigms.

Software Architecture

The many strands of work in software design described thus far have most fruitfully blended and matured into the field of software architecture. With an encompassing definition of software architecture as "the set of principal design decisions governing a system"[78], it engages the full range of design activities and includes the full range of participants in the design process. Software architecture encompasses work in modeling and representation, design methods, analysis, visualization, supporting the realization of designs into code, experience capture and reuse, product lines, deployment and mobility, security, adaptation, and so on.

Software architecture research began in earnest in the early 1990's (e.g. [66, 73]), though the term is decades older (it is found, for instance, in many works from the early 70's). Work in the 90's was initially focused largely on matters of design representation [56], though the whole movement could be characterized by a desire to provide substance, structure, and specificity to the historic field of software design. Arguably, software architecture has been focused on the maturation and professionalization of a field previously best characterized as craftsman-like.

The successes achieved by software architecture over the past fifteen years span from commercial use of product-line architectures, such as at Philips [59], to the architectural underpinnings of the World Wide Web, as characterized by the REST style [27].

Architecture is a backdrop for much of the directions for software design that we characterize in the remainder of the paper; its importance is reflected by its inclusion in this paper's title.

4. Design, Designing, and Designers

The careful reader will notice that we have, so far, refrained from defining software design, or even design itself. So it is with the contributions discussed in Sections 2 and 3, which by and large typify design according to a certain perspective (e.g., a phase, in the code, art, engineering) and then work within this perspective to make their contributions.

To understand how these perspectives relate and together help or hinder in advancing the field as a whole, it is critical that the field establishes a common basis from which its progress can be judged. The following summarizes a general model of design that is intended to form such a basis [13]. The model consists of two interrelated parts: one part capturing the essence of design-*the-product*, the other part capturing the essence of design-*the-process*.

When used as a noun, design normally indicates the artifact (product) that emerges from the design process, some physical document or other kind of representation that articulates the intent of the designer. This product results from the choices the designer made, choices that form an abstraction of that what is eventually desired to be realized in the real world [49].

These words are in some ways obvious, but in other ways not sufficiently precise to help guide a field. The general design literature has made various characterizations that *can* be used as such (e.g., Simon [74], Norman [58], Schön [72]). Figure 2 presents a visual of the amalgamate of these characterizations as they pertain to the design product. The figure distinguishes the design space from the outcome space. During design, we mentally operate in the design space (where each point represents a unique set of design decisions), but continuously make decisions that reflect upon the outcome space (where each point represents a unique artifact). That is, each design decision alters the set of outcomes that are still possible (SP), cutting away some and re-enabling others. *A design, then, is a point in the design space that represents a*

set of decisions that together delineate a set of possible outcomes in the outcome space.[2]

The customer brings into this their understanding of what are desirable outcomes (D), which, whether or not explicitly stated, act as constraints on the design process. Another set of constraints is exercised by the available materials from which an outcome is constructed by following a design's blueprint: a design should describe only outcomes that are feasible (F). A successful design is one that restricts its still possible outcomes to those that are desirable *and* feasible.

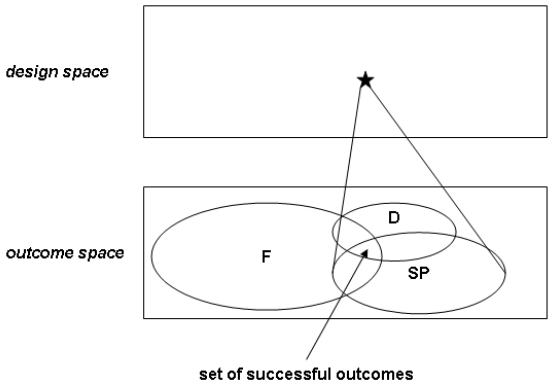


Figure 2. Design – The Product.

When used as a verb, design normally indicates the process by which a design is achieved. It is understood to be a human-centered process, involving varied stakeholders. It is also understood to be strongly goal-driven and drawing upon established knowledge of the designer and the field at large.

The general design literature has made precise characterizations of this process, which are brought together in Figure 3. The design process is one of information manipulation (broadly construed to encompass initial creation, transformation, and deletion), with four types of information involved: goals, ideas, and knowledge, which are all mental, and representations, which are physical expressions of mental information. Each of these types of information is phrased in one or more languages, and tools may be used to edit and/or interpret representations. Within this setting, designers engage in one or more activities, through which they – directly and indirectly – explore the design space. *The design process, then, is defined as the set of information manipulation activities through which a successful design is obtained*.

[2] This discussion is not meant to imply the existence of a separate design *phase.* The *spaces* we refer to are ephemeral and largely present as a result of the way in which human's think – through abstraction.

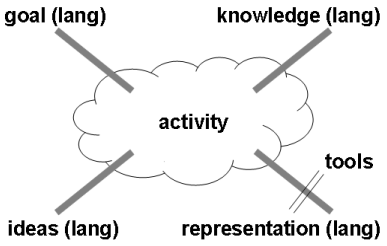


Figure 3. Design – The Process.

An important property of this general model of design is that it does not bias itself towards any individual perspective on software design. Rather, it supports the field in giving it the ability to precisely relate different perspectives and understand their emphases, strengths, and weaknesses.

The Elements of Design Research

A corollary from the preceding discussion is that the model points toward exactly where it is possible for a discipline as a whole to make progress in better supporting its designers. Particularly, it can:

- Improve the *materials* from which a product that is envisioned by a (finalized) design is eventually incarnated in the real world. For example, the availability of newer, lighter composites enabled different kinds of planes exhibiting different weight and aerodynamic properties to be designed.
- Improve the *languages* that are used for capturing goals, ideas, and knowledge. Alexander's design patterns are an example of such an advance, bringing together goals and ideas in a single representation that furthermore supports easy adoption.
- Improve the general *knowledge* the community has about its design domain and design processes. For instance, the human genome project is of tremendous value to the design of new medicines, providing a data bank of knowledge that previously was unattainable.
- Improve the portfolio of *activities* that are used in the design process. IDEO's focused form of brainstorming is an example of an activity that led to improved results, both in terms of time and out-of-the-box solutions [45].
- Improve the *tools* with which design activities are supported, particularly in creating and interpreting representations. For example, the automotive industry has made significant leaps in their ability to design by moving from clay models to CAD/CAM designed 3D visualizations.

All progress, whether in the form of a new methodology, notation, metric, or analysis algorithm, to name a few, will eventually reduce to these five basic underlying categories.

The Community of Designers

Just as it is important to understand the fundamental elements of design, so it is important to recognize the richness of the community of designers – those who design, who contribute to the design, and who must interact with design representations, designers, and design processes. Historically, design has largely been seen as a somewhat provincial task performed by a small number of software specialists – perhaps one "chief designer" – during a circumscribed period in a project's lifecycle, namely following requirements analysis and preceding any implementation. Clearly, such a simplistic notion is either counter to what really happens in a project, or if actually followed yields subpar results. In truth, the number and types of individuals with vital interests in a project's design represents a broad community of interest [29]. Existence of this community imposes some particular demands on design; recognition of the breadth of the community highlights opportunities for improving our practice and expanding our research agenda.

First, for a project of any significant size, more than one designer will be involved. Existence of multiple designers thus imposes demands for communication of design concepts [21]. Communication implies shared representations, an observation that is a natural outflow from the general model of design. The effectiveness of such communication is determined by the language(s) used [22]. Presence of a design team also induces requirements for coordination of its design activities. Such coordination could involve formal management if the task is large enough.

Second, however the design is produced, other individuals, playing other roles, are critically engaged with the design. They must be able to understand and use it. In traditional development practices wherein the implementation activity is separate from the design (more about this below), the implementers must be able to comprehend and utilize the design. The practical challenges of this become highlighted should such implementation be contracted to another firm, perhaps to one on another continent. In other situations, the customer may be desirous of participating in substantive review of a design. In yet other situations end users may be participants in the design process. All of these engagements with design highlight the critical role of and demands on shared representations of design.

Third, beyond just recognizing the existence of a multiplicity of designers and "design readers", we also must recognize that the multiple stakeholders of a system (can) all contribute to the design itself. The typical perspective has been that, since software is being designed, *software* specialists are the only ones who properly determine the software's design. As discussed in detail in [57], however, both application domain experts and business stakeholders are properly contributors to a system's architecture. Domain experts naturally know or determine the key abstractions for a system, acceptable strategies for meeting regulatory requirements, or provide vital insights on what parts and in what ways the design must be flexible to accommodate potential future changes. Business-focused stakeholders may determine key boundaries for a product-line architecture, and hence determine critical software interfaces. Design, thus, is not the exclusive province of the software technologist; the multiple and proper contributions from the wide community of stakeholders must be accommodated. This final comment is not easy to achieve, however. Separate stakeholders likely require (or at least request) views onto an emerging design which are idiosyncratic to their perspectives. Supporting multiple viewpoints while ensuring consistency among them constitutes a current challenge towards which the community has made progress (e.g., the 4+1 model [50], analyses to discover inconsistencies [26], and an understanding that certain forms of inconsistency must be tolerated [15, 28]), but for which much work remains to be done.

Fourth, arguably the design community increasingly includes the end user. Common applications such as spreadsheets and word processors actively invite the end user to customize – i.e., design – their working environment and the complex artifacts produced with desktop tools. At best the software engineering community has barely recognized this community of designers; after all, they are not software specialists, have little or no formal training in design, and produce designs researchers may dismiss as trivial, or simply as poorly done. But the enormous number of end users, the substantial ability for customization and design presented by desktop applications, and the potential for improving the quality of end user designs argues strongly for design researchers to turn their attention to this special world. As one example, spacecraft systems designers use complex, interlocking Excel spreadsheets to design new systems and missions [55]. These complex spreadsheets are designed, however, without any higher level of abstraction or representation; understanding them and "verifying" them is left to the engineer, who must interpret the macros pervading the spreadsheets. A better way is surely possible.

5. Research Directions

The introductory section of this paper indicated how design will remain an enduring challenge. As our ability to design solutions to current statements of wants and needs improves and becomes more predictable, our conception of what might be achieved expands. As "the means" improves and rises, "the task"

becomes more adventuresome and demanding. Our ability to design must be correspondingly enhanced to make use of the improved materials, tools, and mechanisms to create solutions to the new goals and dreams. Seen in this general light, all the elements of design research listed in Section 4 above will persist; it is a matter of understanding and improving:

- the materials – the conceptual building blocks – from which designs are eventually realized as artifacts in the real world.
- the languages that are used for capturing goals, ideas, and knowledge.
- the general knowledge the community has about its design domain and design processes.
- the portfolio of activities that are used in the design process.
- the tools with which design activities are supported.

While comprehensive, this list is too generic to be of much use in setting directions; the remainder of this section is devoted to discussing a variety of more specific directions. Before moving to the more specific discussion however, we consider three general issues.

First is the matter of design decisions. Designing is fundamentally a matter of making choices – how to accomplish something, how to represent something. That suggests that an effective approach to design should offer a solid understanding of choices: what they are, recognizing when they are made, what the alternatives were, and capturing them in such a manner as to allow retrospection. In typical practice we are far from having a grasp on this: decisions are often implicitly made. It is not until later that we recognize that certain functionalities or properties were precluded, or inhibited, by an early choice. Decision support systems from the 1980's, such as gIbis [23], offered explicit support for formally considering choices facing a group. This type of support is needed, but must be provided in a lightweight manner that is integral to and supportive of design. Perhaps more important, however, would be practices to aid in recognizing when choices are being made.

Second is the matter of the place of design in the software engineering process. As the discussion in the previous section indicated, the activity of design is not limited to one individual or one circumscribed place in the development process. Critical decisions – design decisions – are made throughout system development by a variety of stakeholders. Talk of "requirements engineering" that is wholly independent of design, for instance, is frequently either sophistry or simply counter-productive. As critical decisions about a system are made – whenever they are made – design is being done, the architecture is being established and should be recognized as such. Similarly, implementation is improperly and unrealistically considered the rote translation of design to code, sometimes to the point where a design intentionally does not make a choice of programming language so to be general in nature. Key choices made in and about the implementation process, such as the decision to use a particular implementation framework or programming language, are important parts of system design, affecting future strategies for system modification and adaptation. The importance of such design should not be overlooked. The challenge for design researchers is to provide practical guidance to those involved in setting system requirements, in coding the system, and indeed in all other aspects of system development for their appropriate participation in design. More generally, the question is how to structure software development processes to support a robust, modern conception of design.

Third is the matter of choice and evaluation [32, 74]. As designers confront issues, a set of choices emerge. Beyond recognizing and recording the choices made, designers need support for evaluating alternative choices so to guide them towards a design's objectives. Analysis techniques may focus on functional or non-functional system properties, and may span to analysis based on economic arguments [12, 77]. While development of individual techniques continues to be needed, the field also should find ways in which such techniques can be combined to enable multiple properties to be jointly assessed, in the manner of statistical decision theory for example, to enable broadly informed choices to be made.

Directions Reflecting Good Recent Progress

In defining a research agenda, it is important to recognize those research directions that "work", have had impact already, and should be explored further because they continue to have promise in advancing the field.

The first direction we discuss as such is software architecture. It is interesting to put architecture in light of the five directions of design research presented in Section 4. To date, advances have included, among others, architectural middleware, description languages, styles, design methods, and environments, which collectively cover the five research dimensions along which progress can be made. That is, architecture has turned out to be a natural fit in pushing design research forward.

Of particular importance is the focus on early design decisions. Architecture, though it can be mapped onto code effectively, is initially about supporting the exploratory process. Architectural styles are critical in this regard, documenting accepted solution strategies as sets of reusable design decisions that can be readily adopted. Clearly-documented and well-packaged styles

come closest to the original definition of architecture as "structure, form, and rationale" [66].

Architecture has also strongly influenced software product lines. The vast majority of software product lines are actually realized through product line architectures, which are used to distinguish those parts of the system that are shared among all products and those parts that are variable and depend on the product at hand [70]. The use of product lines has become successful with several success stories emerging that detail how this kind of domain-driven approach can be beneficial and provide a competitive advantage.

That said, there is significant work left to do. Design involves multiple stakeholders who may have radically different concerns. Having focused strongly on component-connector centric approaches, current architectural description languages lack facilities for specifying and relating such diverse sets of concerns. Extensible architecture description languages are a foundation, but their modeling capabilities must be supported with flexible environments and design processes.

This strongly relates to the need to manage evolving architectures. When stakeholders make changes, it is often the case that the architecture degenerates. Especially with product line architectures, it is known that even a pungently cohesive initial definition slowly but surely may morph into a set of disjoint product architectures. Processes have been employed to ameliorate this problem, but overall our level of understanding is still limited and our tool support for carrying out such processes trails significantly.

Throughout the design process, whether it is a high-level architecture or a low-level UML class diagram, it is generally important to ensure that certain properties are met, such as, for instance, behavioral consistency, real-time performance constraints, reliability, levels of security, and concurrency behavior. The analysis community has made steady progress in providing analyses that can provide such guarantees and continues to work on faster and more efficient analyses, analyses for new properties, and general infrastructures[16, 24].

A particular challenge is to make these analyses, and the modeling of the information that is needed to drive them, an integral part of the design process, rather than some activity that is performed as a "check" when the design process has finished. Two problems persist: the need to create precise representations, and the need to fully, or almost fully, model a design before it can be analyzed. It is incumbent upon the field that these two problems are overcome, so that analyses become usable throughout and especially when it matters most: during rapid generation and evaluation of (not necessarily precise or complete) alternatives, for which analyses are vital in understanding the tradeoffs inherent in the design decisions made.

As important as the externally-visible qualities of a design are its internal qualities: is its structure sound, is it optimal, and will it hold up over time? A recent trend has seen attempts to assign "value" to designs, particularly by employing economic analyses that stem from adapting and applying established economic theory to the domain of software design. Most promising to date is the use of Design Structure Matrices [14]; with them, it is possible to visually compare different modularizations, assign these modularizations values, and in so doing understand design tradeoffs, such as whether to refactor or to apply aspects. These results, however, represent only a beginning. Values are assigned mostly to entire designs, not necessarily individual design decisions (though these can be valued in the context of given design changes). Values also are "instant", for the design as it is now; not how they stand up against future design changes. A critical dimension of future work, then, is to find mechanisms of valuing individual design decisions over time.

Another important aspect of this work lies in the historical lessons it can teach. Applying Design Structure Matrices to numerous designs, both good and bad, can build a portfolio of examples from which it furthermore may be possible to deduce general principles.

Finally, we return to the topic of architectural styles and, more generally, the capture and reuse of architectural experience. Experience and "good design practice" can be captured at different levels of abstraction (from source code up to the highest level of system structure) and with different degrees of generality (from useful within all application domains to useful only within highly specialized application domains). We need the ability to capture the lessons from prior developments at all points in this space, and to do so in a manner that effectively enables other engineers to find, understand, assess, and apply the lessons to the development of new systems. In simplest terms this could involve developing extensive catalogues of architectural styles. The richness and variety of experience demands better ways of capturing, finding, and using knowledge, however. Other design disciplines have matured by doing so, it is time we do so as well.

Directions From New Capabilities

Progress in computer science has often been propelled – or compelled – forward as the result of advances in hardware. So it is now with design. Advances in networking, display technology, storage, and processor speed offer the potential for significant advances in the practice of design.

As a first instance, consider the potential of searching for domain knowledge, prior designs, evaluations of designs, "similar structures", and so on, in the manner of Google. That is, the ability to access information across the Internet, and especially the ability to

search that information in comprehensive fashion, offers the potential for exploiting experience from prior designs in a manner far beyond anything we have yet seen. A simple use of existing Google-like search will not be sufficient, however. A designer will seldom be searching, for example, for a module with a specific textual interface. A designer will want to search based on various architectural abstractions. Enabling search based on architectural meta-data, for instance, is a near-term possibility for meeting this goal. Yet any search scheme that requires some structured meta-data as input stands the risk of being overtaken by a technology that employs a brute-force strategy that is able to provide at least as good results without requiring use of any standard mark-up or meta-data. This, of course, is the beauty of today's Google search as applied to document searches. One long-term direction, therefore, is to develop search algorithms that perform architectural abstraction automatically, and then "page-rank" those abstractions against the user's query, where that query is phrased in terms of architectural properties.

The networking that is a key enabler of Internet search is also a key enabler of improved communication between individuals and teams. As the legitimate role of the many stakeholders in a design process is recognized, advances in network communications can be brought to bear to improve their participation. Collaboration technologies in general offer significant potential for the design process [39]. Communication technologies are, relatively-speaking, free; designers should exploit that.

Display technology offers another basis for improvement in design practice. Very high resolution, very large screen displays are now readily available. Such displays give designers the potential of seeing more of a design from more perspectives simultaneously. And why should design be confined to the flatland of 2-D displays? Other scientific disciplines have already exploited high-resolution displays; it remains for software designers to design such support for their own discipline. Designers should have displays on their desktops that are at least as large as the televisions in their homes. Beyond the desktop, there is no reason why teams do not have specialized design rooms equipped with numerous touch-sensitive displays and batteries of computers that enable instant analysis.

The continuing decline in the price of storage with an accompanying increase in capacity suggests other new directions for design. Why not always record design rationale – even as video? Keying video/audio capture to the designers workstation activities, context, and display offers unprecedented potential for retrospectively understanding a design and reusing the insights present at the time of design. In the case of design forensics following a system failure, for example, such storage offers the potential for identifying the root cause of a failure, and hence for eliminating related latent errors elsewhere in a system.

Lastly, the continuing rise in processor speeds suggests that designers, and those who develop design tools, should never be restrained by a perception of something "taking too long". Nor should tool developers be distracted into complicated optimizations of, e.g., analysis procedures, when the use of simple brute force suffices. If something seems to take too long, just task a few dozen more processors to the problem, or simply wait for the next generation of processors to appear. The time to develop a reliable optimization may well exceed the time for a doubling of processor speed.

In summary, let us as designers exploit advances in computation to advance the practice of design. Our task is as worthy of innovative use of technologies as our clients' tasks are.

Directions From Design Imperatives

For software design and architecture to mature into a robust discipline capable of handling the challenges posed by emerging applications, advances in several areas are imperative. Our list here is eclectic, reflecting our perception of particularly poignant needs.

First is adequate support for moving from architecture to implementation, and fluidly moving between design and coding tasks. If design does not ultimately support production of a satisfactory implementation (assuming that the resources and the will to produce are both present), then it is a failed effort. Many current design approaches fall silent when it comes to implementation. Since key design decisions may be made in the implementation context, evolution of the architecture must be seamlessly integrated across the contexts. Seamless integration implies full traceability between code and higher abstractions, and supports accountability of design decisions.

Second is the ability to represent all stakeholders interests, as discussed earlier. Multiple interests and multiple perspectives impose demands for assessing or guaranteeing consistency of design decisions (or for the management of inconsistencies between them), as well as demands for multiple presentations of select design data.

Third, as software applications become ever-more interwoven into organizations and society, we must develop means for the co-design of software *and* organizational/societal systems. Introduction of a technology into a group or organization may radically change how the group behaves – think, for example, of how e-mail has altered both personal and public communication patterns. While a robust literature exists describing how such organizational changes have occurred as the result of introducing software technologies, we need a design discipline which integrates the inten-

tional shaping of software technology with the intentional shaping of organizations (one easy example is the integrated performance of business process reengineering with design of software systems for that business). For co-design to take place, a broad range of expertise must be woven into the process of design. To continue without such breadth invites organizational "surprises" and application system failures.

Fourth is the design of applications as seen and experienced by users. This was discussed in the Introduction as the "second type" of design. The need and opportunity is profound; further discussion is reserved for the Challenges section.

The next two directions are closely related and are motivated, at least in part, by economics. The first of these is supporting design recovery and analysis. Established systems represent significant economic assets. To the extent that such assets can be used to meet new organizational needs, economic efficiencies are realized. Recovering the architecture of existing systems enables assessments of potential future uses to be made so that adaptations can be based on solid architectural understandings. While several research projects in this field exist and have yielded promising initial results (see, e.g., [18, 36, 37]), there is still much to be done. The second and related direction is actively managing design evolution – in particular mitigating architectural decay. Here the issue is not recovering a design to merely enable the first steps of progress to a new or improved system, but the task of assessing an existing design and evaluating alternatives for modifying it to meet new and changing needs. Clearly to the extent the architecture is explicit and faithful to the implementation, this process is facilitated. But that is only the beginning; an evaluation framework and process to assist in comparing alternative modifications is needed. Such evaluations must not only support examining how immediate demands can be met, but predict likely consequences for future, as yet unspecified, demands.

Lastly, emerging application needs argue for design techniques that yield self-adaptive systems. An important topic in its own right, we refer the reader to [48].

Directions From Examining Our Past

We learn from our successes, but we also learn from our mistakes. This is an age-old lesson that has fueled much progress in other design fields. Bridge design as it is today would not be as advanced without the careful study of past structural failures [76]. The study of "why things break" has fueled the creation of newer, stronger materials [25]. And a central theme in Petroski's well-known writings is how failure has been a motivator for design innovation [67-69].

How do we perform in software in this regard? Unfortunately, the answer is "not good". We do experience failures, but the field does not profit from them as other disciplines do. A "we will just fix it in the code" attitude is far too prevalent, and we seem to have been lulled into a modus operandus in which the importance of design is, consciously and subconsciously, undervalued. Compare the software view of design once again with that of bridge design. First off, one must appreciate the effort that goes into a bridge's design. Except for a few systems, our discipline rarely performs this much design. Second, when something fails on a bridge, it is the design that is examined, and lessons are drawn from it. Such is not the practice in software; we rarely go back, carefully study a "failed design", deduce lessons as to why the software (use, deployment, or even development itself) failed, and what we should do differently, design-wise, next time – let alone share these lessons with the community.

Any approach to learning from the past must start with examples. Unfortunately, no software design examples seem to be available. Textbooks contain smallish systems. Search the web for "Good Software Design" or "Bad Software Design Examples" and not a single system comes forward (though lots of advice on how to create a "good" design comes forward). Compare this to building architecture, where one can find book after book in the bookstore, including books of "great designs". Clearly, a first challenge for the community is to begin assembling archives of good and bad design examples. By this we do not just mean UML diagrams, but carefully documented, multi-level and multi-view explanations that provide in-depth insight into underlying design decisions and their ramifications. It is interesting, in this regard, to examine the HCI literature. In it, one can find numerous papers that introduce novel interfaces and describe their underlying design motivations. Such a practice has not transitioned into the software engineering conferences as of yet.

We must also promote excellence in design. The SPLC product line hall of fame is an example of such recognition, as is the aforementioned ACM Software Systems Award. In either case, though, we note that regrettably the actual details of the product and design that received the award are never shared with the broader community. Perhaps ACM or IEEE should consider establishing an annual prize for the best software design, requiring that winning designs are placed in the public domain.[3]

From examples, we must then deduce patterns, principles, do's and do not's, and other general understandings that help individuals in building up a repertoire of knowledge that can assist them in designing. Some of that effort is underway; we mention architectural styles, software patterns and anti-patterns, design

[3] Of course, we can leave our version of the Razzies (http://www.razzies.com) to an appropriate blog.

critics, bad smell detection and refactoring techniques, HCI design guidelines, and a small handful of general design principles. This work has to continue and be broadened to cover all aspects of the design product and process.

Finally, we observe that we must not just understand good and bad design products, but also focus some of our efforts on understanding good and bad design practices. By this we explicitly do not mean high-level approaches (e.g., Agile), but rather the approaches and techniques that expert designers employ in designing their software. For instance, it is well-known that product designers may sketch hundreds of alternatives before honing in on an eventual choice. Do software design experts follow such an approach? If not, what do they do that makes them successful? Can these skills be explicated and communicated to others?

Overall, the undercurrent of this section is that software design is still far removed from being an established discipline. To move forward, it is critical the field engages in the necessary deep scientific study of software design and designing.

Directions From Looking Outside of CS

While the design of software is a relatively new activity, having only been around for sixty years at the most, design has been practiced in other fields for centuries. While sometimes still taught as a craft, or learned through apprenticeship, design is newly taking shape as an academic discipline, even as a science. There is a Design Research Society [1] (which sponsors a conference on doctorate research in design), and a wide literature. New university programs in design are emerging, such as Stanford's "d-School" [7]. All this suggests that there is much that software researchers can consider and draw from in order to advance design specifically within the software field. A few examples have already appeared in the preceding text: we have referred to work in industrial engineering [44], architecture [11, 19], design processes [72], and civil/mechanical engineering. We provide a few additional examples here, most of which are inspired by building architecture.

While architecture has already been mentioned, and has been used for many years as, at least, an analogous activity to building software, the richness of the building architecture discipline suggests that there are still further insights to mine. For instance, Parnas's dictum about designing software for ease of change [63] is explored, by analogy, with substantially greater richness in Stewart Brand's "How Buildings Learn" [19]. The several layers of a building's architecture determine the ways in which the building can be adapted to meet new needs. Bottom-up and top-down approaches to such design are considered and extensively illustrated. It is a book that, while containing no reference to software development, can be read, appreciated, and applied by software engineers.

Another practice from architecture is that of a design charette. A charette is related to software's design reviews and walkthroughs, but is closer in spirit to agile design, for the purpose of a charette is to move a design forward quickly, by developing and critiquing design in a group setting. In educational settings, charettes are part of design studios, where regular design reviews take place. The normal practice of architecture is to develop models suitable for and used in periodic, active, productive, constructive group design reviews.

Perhaps most inspirational from the world of architectural design is the development of computer-based building models that enable designers and users (tenants, residents) alike to fly through a proposed design, simultaneously seeing, as desired, both the internal structure of the building and the appearance and services of the building. In software design these concepts are almost always reviewed separately: how the code is organized is considered almost independently of what the user's experience with the application will be. In building architecture, the intrinsic relationship between these two views is understood; if the residents of a house know in advance where load-bearing beams are they can adjust their expectations for how the building might be modified in the future. Seeing how well, or how poorly, the user interface is separated from the rest of an application's code can reveal whether managers could sensibly direct a desktop application to be retargeted as a web service.

Architecture even suggests how we might rethink the composition of our academic teaching faculty. Architecture schools typically include many practicing architects, just as music schools include many practicing musicians. The conviction is that students can only have an adequate understanding of their discipline through engagement with faculty regularly acquainted with the full breadth of disciplinary challenges. One could examine many computer science departments and find no faculty qualified to construct any application larger than a compiler – a task trivial in comparison with the challenges regularly faced by many professionals in industry. Students must also engage in the practice, and do so repeatedly.

Many design professions have another practice from which we could benefit: the study of designs. Designers of luxury goods, buildings, machinery, and consumer goods alike spend significant time assessing – studying – existing designs. The objective is not only to understand how something works, but to assess its "non-functional properties" – what brand sense it conveys, what its aesthetic is, how it affects its user. In contrast, most software engineering courses spend no time studying existing design, instead plunging ahead

with green-field approaches, yielding a too-predictable outcome.

Barriers To Progress

Having made a broad set of suggestions, we must acknowledge that significant barriers persist to making these kinds of advances a reality in practice. First, as we observed earlier, design is seriously undervalued by many. On the one hand, we admit that there is some basis for such undervaluing: the tools and techniques available to the average practitioner are not necessarily very good, especially when put in light of the full spectrum of considerations discussed in Section 4. This state of affairs makes it difficult to convert the skeptic or to provide credible evidence that proper design(ing) does make a difference. On the other hand, such negative attitude hinders progress: the effort spent objecting might be better spent making advances or at least encouraging others to do so.

Second, designers are not necessarily equipped with the right skills and, worse, they may or may not know whether their skills match up to a project at hand. Who is qualified and how do they acquire their skills? Certainly, some designers are simply great, whether by experience or intuition. But a vast majority has to acquire their skills somehow, yet a culture of apprenticeship is virtually non-existent. Granted, not all software needs a great designer, but even the "average designer" must learn somehow.

Compounding this problem is the remarkable tolerance that software professionals seem to have. If the tools that we use to design are incomplete, inelegant, and difficult in their use, then how can we be expected to produce designs that are complete, elegant, and lead to easy-to-use systems? And this does not just hold for design tools. Poorly designed user interfaces and clunky programs abound. Where are we to find our inspiration for quality?

A root cause can be found in the education of software engineers [49, 53]. Most stem from a "standard" Computer Science program, which incorporates *at best* a few software engineering courses and involves numerous other courses which ignore the lessons of software engineering altogether. Extensive practice with significant software design problems is impossible in this setting. The past decade has seen the emergence of Software Engineering (e.g, [5, 6]) and Informatics (e.g., [2-4]) majors. The focus of most such programs is on design, a trend we welcome. Still, the materials available from which to teach design are limited and much innovation is necessary in this regard.

To offer hope, there is the advantage of time. Early reports from the SIGSOFT Impact project indicate that research advances may take up to ten or sometimes even twenty years from initial idea to widespread practical use, as the original idea must find traction and morph to reflect practical needs and considerations [60]. So, in a field that is as young as software engineering, perhaps we are not doing so badly?

Improvements and overcoming the barriers we mentioned, though, will require the community to undergo a drastic change in mindset. Rather than following the next hype into believing software development "can be made easy", a true discipline must emerge in which it is recognized that design is a critical activity that involves serious and difficult work. And herein may lay the most difficult barrier of them all.

6. Challenges and Vision

Design and architecture as described comprises a broad field and arguably sits at the very core of software engineering. All of its aspects are vital: ways of designing, architectural representations, means for performing analysis, techniques for transitioning a design into an implementation, ways of capturing design experience, and so on. Absence of progress in any one of the areas discussed impedes progress in the others. Thus, broadly based advancement on the whole set of sub-topics constitutes a grand challenge for software engineering – an appropriate, critical focus for software engineering research. We conclude, therefore, not with a specific design problem as a grand challenge for the field, but rather repeat and highlight a few technical items, providing a bit of a vision for the future, offer some directions for community activities, and finish with a speculation on the long future of "software" design.

Technical Challenges

Designing a software application involves designing its structure as well as its user-observable properties, functional and non-functional alike. By removing the counterproductive boundary between requirements and design, a holistic view of product conception emerges. By analogy to building architecture, a building can be seen as being composed of beams, bricks, pipes, glass, and wires, but also being composed of living spaces, galleries, sun rooms, and cooking facilities. Building architects can show clients cut-away views of buildings, simultaneously revealing both structure and facility, the interrelationships between natural light and ceiling truss. Imagine now interactive cut-aways of buildings, whereby the designer could move skylights and see the consequences for the roof's structure, or change specifications on a window and note how the quality of interior light improves.

Realizing an analogous vision for software design requires our supporting the design and visualization of user functionality at least as well as our supporting the design and visualization of software structures. Not only must we see and manipulate components and

connectors in an architecture, but see, as we do so, how the user's data display is changed, or how the electronic commerce purchasing experience is shaped, or how facilities for controlling the chemical plant are set.

The community must develop the languages, techniques, and tools for enabling the multitude of stakeholders in an application design to sketch, evaluate, revise, and refine design concepts for applications. Success will be achieved when clients are able, through working with the design team, to see their tasks in new ways and are able to innovate new ways of meeting those tasks.

Community Challenges

Achieving the technical vision will require long-term financial support, new venues for publication and community analysis of designs and design technology, a new sense of who the design community comprises, and, perhaps, some "incentives".

The critical enabler for progress is, of course, adequate funding. The centrality of design and architecture to software development demands that significant, stable funding be directed to these activities. The National Science Foundation's Science of Design program is a good start in this direction, but support for design research should not be limited to the NSF; other agencies should make this field a priority as well. Such support should also be *continuing*. Design will not be "solved" after three years of work; indeed, as we have argued, design will always be a challenge – our aspirations will not abate.

Software design and architecture research also needs adequate forums for the presentation and review of research advances. Drawing from the study of design in other fields, forums are also needed for the public presentation and review of designs themselves. Such review can inspire other designers, reveal properties of new design techniques and tools, and add to the repertoire of design experience. Design research often does not have the same character as research in other fields, such as software testing and analysis. Hence traditional forums and criteria are unlikely to be adequate or appropriate for review of design advances.

New forums for the discussion of software design would also be supportive of expanding the community of contributors. As software design is recognized as engaging teams of designers with expertise spanning specific application domains, business planners, and software specialists, a forum in which all would feel "at home" would be productive.

Lastly, there is nothing like an incentive to spark quick advances in a field. Building architects are annually awarded the Pritzker Prize, an award that carries not only international fame but a $100,000 reward. In fact, design awards are common in many fields: automotive design, industrial design, fashion design, and so on. Why not spark innovation in software design by creation of a corresponding prize? A similar kind of inducement for advancement is a challenge prize, such as the Ansari X-Prize was for space flight. Establishing appropriate criteria and processes for evaluating software designs would be a challenge, but the effect on the community could be significant. For instance, the evaluation could cover the process by which the design was produced, as well as design itself.

A Vision For The Long Future

One of the themes of this article has been that software design and architecture has been, and will remain, an intellectual challenge: as our abilities to effectively design for one set of challenges become more effective, a new set of design challenges emerge to demand yet further advances. The limit on this cycle is simply the definition of "software". By expanding our sense of what software is, in a very liberal sense, numerous exciting opportunities for contributions from software design researchers emerge, opportunities for which we, as software designers, have some distinct advantages over designers from many other fields.

Consider, for example, the interaction design problem faced by automotive designers. One could argue that what car manufacturers are selling is not sheet metal and rubber, but a "driving experience". Such an experience constitutes a structured amalgam of sights, sounds, feelings, and smells. Driving a car involves interaction with not only the control devices in the car, but interaction with passengers, audio and visual inputs, interaction with other vehicles, laws, and traffic control systems outside the vehicle. If the design problem is to design *that* interaction experience, what representation does that design have? The interaction is fundamentally intangible. Traditional design disciplines, such as automotive design, are strongly grounded in the physical materials of their traditional products and designers are unaccustomed to a final reality that is intangible. Software designers, by contrast, have always dealt with an intangible product: software; we are accustomed to designing, modeling, and assessing a broader set of realities.

If we therefore extrapolate the concept of "software" beyond the traditional confines of the computer to encompass radically different intangible products, such as this example automotive interaction, an exciting frontier opens up. Software designers may be able to lead the way into conceptualizing, modeling, and building new kinds of intangible products. Working cooperatively with designers from other specialties, the prospect is for creating new kinds of highly complex systems that are now barely imaginable.

Software design and architecture have a long future ahead of it.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE

7. Acknowledgments

This work was supported in part by the National Science Foundation, under grants 0438996 and 0536203.

We would like to thank Alex Baker, Eric Dashofy, Peter Freeman, Michael Gorlick, Neno Medvidovic, Peyman Oreizy, David Redmiles, Lee Osterweil, and Alexander Wolf for ongoing collaborations and fruitful discussions that have fueled the opinions expressed in this paper.

8. References

[1] *Design Research Society*. <http://www.designresearchsociety.org/>.

[2] *University of California, Irvine, Donald Bren School of Information and Computer Sciences, B.S. in Informatics*. <http://www.ics.uci.edu/informatics>.

[3] *Indiana University School of Informatics, B.S. of Informatics*. <http://www.informatics.indiana.edu/>.

[4] *University of Washington Information School, B.S. of Science in Informatics*. <http://www.ischool.washington.edu>.

[5] *Milwaukee School of Engineering, B.S. in Software Engineering*. <http://www.msoe.edu/eecs/se/>.

[6] *Rochester Institute of Technology Department of Software Engineering, B.S in Software Engineering*. <http://www.se.rit.edu/degrees.html>.

[7] *d.school – The Hasso Plattner Institute of Design at Stanford*. <http://www.stanford.edu/group/dschool/>.

[8] Abbott, R.J. Program Design by Informal English Descriptions. *Communications of the ACM*. 26(11), p. 882-894, 1983.

[9] Abowd, G., Allen, R., and Garlan, D. Using Style to Understand Descriptions of Software Architecture. *ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*. p. 9-20, ACM Press. Redondo Beach, CA, December, 1993.

[10] Agrawala, A., Krause, J., and Vestal, S. Domain-specific software architectures for intelligent guidance, navigation and control. *1992 IEEE Symposium on Computer-Aided Control System Design*. p. 110-116, March, 1992.

[11] Alexander, C. *The Timeless Way of Building*. Oxford University Press: New York, 1979.

[12] Bajcharya, S., Ngo, T., and Lopes, C.V. On Using Net Options Value as a Value Based Design Framework. *Seventh International Workshop on Economics-Driven Software Engineering Research at ICSE'05*. May, 2005.

[13] Baker, A. and van der Hoek, A. *Examining Software Design from a General Design Perspective*. Institute for Software Research, University of California, Irvine, Technical Report UCI-ISR-06-15, October, 2006.

[14] Baldwin, C.Y. and Clark, K.B. *Design Rules: The Power of Modularity*. 1, MIT Press: Cambridge, Mass., 2000.

[15] Balzer, R. Tolerating Inconsistency. *Thirteenth International Conference on Software Engineering*. p. 158-165, IEEE Computer Society Press. Austin, Texas, 1991.

[16] Binkley, D. Source Code Analysis: A Road Map. *Future of Software Engineering 2007* Briand, L. and Wolf, A. eds. IEEE-CS Pres, 2007.

[17] Booch, G. Object-Oriented Development. *IEEE TSE*. 12(2), p. 211-221, 1986.

[18] Bowman, I.T., Holt, R.C., and Brewster, N.V. Linux as a Case Study: Its Extracted Software Architecture. *Twenty-first International Conference on Software Engineering*. Los Angeles, May 16-22, 1999.

[19] Brand, S. *How Buildings Learn: What Happens After They're Built*. Penguin Books, 1994.

[20] Brooks, F.P. *The Mythical Man-Month: Essays on Software Engineering*. 2 ed., Addison-Wesley, 1995.

[21] Brooks Jr., F.P. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[22] Clark, H. and Brennan, S. Grounding in Communication. *Perspectives on Socially Shared Cognition*. American Psychological Association, 1991.

[23] Conklin, J. and Begeman, M.L. gIBIS: A Hypertext Tool for Exploratory Policy Discussion. *ACM Transactions on Information Systems*: 6(4), p. 303-331 1988.

[24] Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, and Visser, W. Formal Software Analysis: Emerging Trends in Software Model Checking. *Future of Software Engineering 2007* Briand, L. and Wolf, A. eds. IEEE-CS Press, 2007.

[25] Eberhart, M. *Why Things Break: Understanding the World by the Way It Comes Apart*. Harmony Books: New York, 2003.

[26] Egyed, A. Consistent Adaptation and Evolution of Class Diagrams during Refinement. *Seventh International Conference on Fundamental Approaches to Software Engineering*. p. 37-53, Barcelona, Spain, 2005.

[27] Fielding, R.T. and Taylor, R.N. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology*. 2(2), p. 115-150, May, 2002.

[28] Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. Inconsistency Handling in Multi-perspective Specifications. *IEEE TSE*. 20(8), p. 569-578, 1993.

[29] Fischer, G. Communities of Interest: Learning through the Interaction of Multiple Knowledge Systems. *User Modeling*. 2001.

[30] France, R. and Rumpe, B. Model-driven Development of Complex Systems: A Research Roadmap. *Future of Software Engineering 2007* Briand, L. and Wolf, A. eds. IEEE-CS Press, 2007.

[31] Freeman, P. The Central Role of Design in Software Engineering. *Software Engineering Education* Freeman, P. and Wasserman, A. eds. Springer-Verlag: New York, 1976.

[32] Freeman, P. The Central Role of Design in Software Engineering: Implications for Research. In *Software Engineering: Research Directions*. p. 121-132, Academic Press, 1980.

[33] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-*

Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley: Reading, MA, 1995.

[34] Garlan, D., Allen, R., and Ockerbloom, J. Exploiting Style in Architectural Design Environments. *ACM SIGSOFT '94 Second Symposium on the Foundations of Software Engineering*. p. 175-188, ACM Press. New Orleans, LA, December, 1994..

[35] Greenbaum, J. and Madsen, K.H. Small Changes: Starting a Participatory Design Process by Giving Participants a Voice. *Participatory Design: Principles and Practices* Schuler, D. and Namioka, A. eds. Lawrence Erlbaum Associates: Hillsdale, New Jersey, 1993.

[36] Gröne, B., Knöpfel, A., and Kugel, R. Architecture recovery of Apache 1.3 -- A case study. *2002 International Conference on Software Engineering Research and Practice*. Las Vegas, 2002.

[37] Hassan, A.E. and Holt, R.C. Architecture recovery of web applications. *Twenty-fourth International Conference on Software Engineering*. May, 2002.

[38] Hayes-Roth, B., Pfleger, K., Lalanda, P., Morignot, P., and Balabanovic, M. A domain-specific software architecture for adaptive intelligent systems. *IEEE TSE*. 21(4), p. 288-301, April, 1995.

[39] Herbsleb, J. Global Software Engineering: The Future of Socio-technical Coordination. *Future of Software Engineering 2007* Briand, L. and Wolf, A. eds. IEEE-CS Press, 2007.

[40] Issarny, V., Caporuscio, M., and Georgantas, N. A Perspective on the Future of Middleware-Based Software Engineering *Future of Software Engineering 2007* Briand, L. and Wolf, A. eds. IEEE-CS Press, 2007.

[41] Jackson, M. *System Development*. Prentice Hall: Englewood Cliffs, N.J., 1983.

[42] Jackson, M. *Problem Frames*. Addison-Wesley Professional: Reading, MA, 2001.

[43] Jackson, M.A. *Principles of Program Design*. Academic Press, 1975.

[44] Jones, J.C. *Design Methods: Seeds of Human Futures*. John Wiley & Sons, Ltd.: New York, 1970.

[45] Kelley, T., Littman, J., and Peters, T. *The Art of Innovation: Lessons in Creativity from IDEO, America's Leading Design Firm*. Currency/Doubleday: New York, 2001.

[46] Knuth, D.E. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

[47] Knuth, D.E. *Literate Programming*. CSLI Lecture Notes, no. 27., Stanford, California: Center for the Study of Language and Information, 1992.

[48] Kramer, J. and Magee, J. Self-Managed Systems: An Architectural Challenge *Future of Software Engineering 2007* Briand, L. and Wolf, A. eds. IEEE-CS Press, 2007.

[49] Kramer, J. Is Abstraction the Key to Computing? *Communications of the ACM*. 2007. To appear.

[50] Kruchten, P. The 4+1 View Model of Architecture. *IEEE Software*. 12(6), p. 42-50, November, 1995.

[51] Kruchten, P. *The Rational Unified Process: An Introduction*. Addison-Wesley: Reading, MA, 2000.

[52] Larman, C. *Applying UML and Patterns. An introduction to object-oriented analysis and design and the Unified Process*. 2nd ed. Prentice-Hall PTR, 2002.

[53] Lethbridge, T., Diaz-Herrera, J., LeBlanc, R., and Thompson, J. Improving Software Practice through Education: Challenges and Future Trends *Future of Software Engineering 2007* Briand, L. and Wolf, A. eds. IEEE-CS Press, 2007.

[54] Lopes, C.V., Kiczales, G., Mendhekar, A., Maeda, C., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. *European Conference on Object-Oriented Programming*. Finland, 1997.

[55] Mark, G., Abrams, S., and Nassif, N. Group-to-Group Distance Collaboration: Examining the "Space Between". *Eighth European Conference of Computer-Supported Cooperative Work*. p. 99-118, Helsinki, Finland, September 14-18, 2003.

[56] Medvidovic, N. and Taylor, R.N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE TSE*. 26(1), p. 70-93, January, 2000.

[57] Medvidovic, N., Dashofy, E., and Taylor, R.N. Moving Architectural Description from Under the Technology Lamppost. *Information and Software Technology*. 49(1), p. 12-31, 2007.

[58] Norman, D.A. *The Design of Everyday Things*. 1st Basic paperback ed., Basic Books: New York, 2002.

[59] Ommering, R.v., Linden, F.v.d., Kramer, J., and Magee, J. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*. 33(3), p. 78-85, March, 2000.

[60] Osterweil, L., Ghezzi, C., Kramer, J., and Wolf, A. Editorial *ACM TOSEM*. 14(4), p. 381-382, 2005.

[61] Parnas, D.L. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*. 15(12), p. 1053-1058, 1972.

[62] Parnas, D.L. On the Design and Development of Program Families. *IEEE TSE*. 2(1), p. 1-9, 1976.

[63] Parnas, D.L. Designing Software for Ease of Extension and Contraction. *IEEE TSE*. 5(2), p. 128-137, 1979.

[64] Parnas, D.L., Clements, P.C., and Weiss, D.M. The Modular Structure of Complex Systems. *IEEE TSE*. 11(3), p. 259-266, March, 1985.

[65] Parnas, D.L. and Clements, P.C. A Rational Design Process: How and Why to Fake It. *IEEE TSE*. 12(2), p. 251-257, February, 1986.

[66] Perry, D.E. and Wolf, A.L. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*. 17(4), p. 40-52, October, 1992.

[67] Petroski, H. *To Engineer is Human*. St. Martin's Press 1985.

[68] Petroski, H. *The Evolution of Useful Things*. Alfred A. Knopf, Inc., 1992.

[69] Petroski, H. *Invention by Design: How engineers get from thought to thing*. Harvard University Press, 1996.

[70] Pohl, K., Böckle, G., and van der Linden, F.J. *Software Product Line Engineering: Foundations, Principles and Techniques*. 1 ed., Springer, 2005.

[71] Sarkar, S., Rama, G.M., and Kak, A.C. API-Based and Information-Theoretic Metrics for Measuring the Quality of Software Modularization. *IEEE TSE*. 33(1), p. 14-32, January, 2007.

[72] Schön, D. *The Reflective Practitioner: How Professionals Think in Action*., Basic Books, Inc. Publishers: New York, 1983.
[73] Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., and Zelesnik, G. Abstractions for Software Architecture and Tools to Support Them. *IEEE TSE.* 21(4), p. 314-335, April, 1995.
[74] Simon, H.A. *The Sciences of the Artificial*. 1st ed. The MIT Press, 1969.
[75] Simon, H.A. *The Sciences of the Artificial*. 2nd ed. The MIT Press, 1981.
[76] Spector, A. and Gifford, D. A Computer Science Perspective of Bridge Design. *Communications of the ACM.* 29(4), p. 267-283, April, 1986.
[77] Sullivan, K.J., Chalasani, P., Jha, S., and Sazawal, V. Software Design as an Investment Activity: A Real Options Perspective. *Real Options and Business Strategy: Applications to Decision Making* Trigeorgis, L. ed. Risk Books, 1999.
[78] Taylor, R.N., Medvidovic, N., and Dashofy, E.M. *Software Architecture: Foundations, Theory, and* Practice. John Wiley & Sons, 2008. In press.
[79] Tracz, W. DSSA (Domain-Specific Software Architecture): Pedagogical Example. *ACM SIGSOFT Software Engineering Notes.* 20(3), July, 1995.
[80] Van Duyne, D.K., Landay, J.A., and Hong, J.I. *The Design of Sites : Patterns, Principles, and Processes for Crafting a Customer-centered Web Experience*. Addison-Wesley: Boston, 2003.
[81] Yourdon, E. *Techniques of Program Structure and Design*. Prentice-Hall: Englewood Cliffs, N.J., 1975.
[82] Yourdon, E. and Constantine, L.L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1979.

Future of Software Engineering(FOSE'07)
0-7695-2829-5/07 $20.00 © 2007 IEEE