Int. J. Engng Ed. Vol. 22, No. 3, pp. 540–550, 2006
Printed in Great Britain.

0949-149X/91 $3.00+0.00

Is Designing Software Different From Designing Other Things?*

DAVID SOCHA
Center for Urban Simulation and Policy Analysis, University of Washington, Seattle, WA 98195, USA. E-mail: socha@cs.washington.edu

SKIP WALTER
Chief Technology Officer, Attenex Corporation, Seattle, WA 98104, USA. E-mail: skip@attenex.com

This paper explores the question of whether designing software is different from designing other things (we believe it is). We discuss several key distinctions that are largely missing from the discourse on software design yet which are vital to the success of software designs. These distinctions are increasingly important as software becomes prevalent in the design tools and products of other engineering disciplines. By considering what is similar and what is different we help reveal how the lessons of software design may help other disciplines, and vice versa. This in turn illuminates a core meta-question of how educators in academia and industry can help evolve our understanding of what we do so that we can be more effective at software design. But first, we need to understand what is different, and what is not different, about this discipline called software design.

Keywords: software design; architecture; complex adaptive systems; design; design process; digital artifacts; organizational design; organizational intervention; pattern language; test-driven development.

INTRODUCTION

THIS PAPER is part of our exploration into how to improve the profession of software development. In our 60+ years of experience in developing, using, managing and teaching about all forms of software and software development, we wondered why there are few examples of great software designs. We wondered why there are so few good software designers among the 2.5 million professional programmers in the United States. We see few resources in the form of books, seminars and college courses focused on designing software. We find very little research that looks at how other fields of design may help us do better software design, or that discusses whether software design is so different that it cannot draw from other fields of design. Thus, our exploration of the question posed in the title of this paper 'Is designing software different from designing other things?'

We believe that software design is different. In this paper we identify major differences between software design and other forms of design, whether these are 'hard design' (for material objects) or 'soft design' (for processes or policies). We will not distinguish whether software design should be seen as a type of soft design. Our main point is that it is different from all other forms of hard and soft design. Understanding whether designing software is different from designing other things will help our software discipline learn from other disciplines, and contribute to other design disciplines. We believe that this understanding is critical, since most design fields are seeing an increasing role for software as aids to the creation of the design, or of software as part of making a more interactive end product.

John Heskett starts off his graduate course on the Economics of Design with the nonsensical statement [1] '*Design is to design a design to produce a design!*' He follows this slide with a partial list of the many different fields of design—engineering design, product design, industrial design, ceramic design, decorative design, graphic design, illustration design, information design, typographic design, advertising design, packaging design, brand design, interior design, pattern design, software design, systems design, interaction design, hair design . . .

The multiple uses of the word 'design' make it difficult to identify similarities and differences in the diverse fields of design. Parsing the sentence to make sense of it we get:

'Design is to design a design to produce a design.'
Noun 1; Verb; Noun 2; Noun 3.

Heskett's working definitions of these uses of 'design' are [1]:

- *Noun 1*—indicating a general concept of a field as a whole.
  - Example: 'Design is important to the national economy.'

* Accepted 12 December 2005.

- *Verb*—indicating action or process.
  - Example: 'She is commissioned to design a new kitchen blender.'
- *Noun 2*—signifying a concept or proposal.
  - Example: 'The design was presented to the client for approval.'
- *Noun 3*—indicating a finished product of some kind, the concept made actual.
  - Example: 'The new VW Beetle revives a classic design.'

At this level of abstraction, the definitions apply to each field of hard and soft design. It is the working practice in each of these definitions which differentiates software design from other forms of design.

We believe that software design is different because:

- the activities that make up the software design field (Noun 1), including organizational design and intervention, span more expertise and domain boundaries than do other forms of design;
- the optimum software design process is the reverse of the process for hard object or product design (Verb);
- the design specification is the software source code (Noun 2) which is complete enough to generate automatically an end product (Noun 3) without translation into humanly understood forms;
- the finished outcome of software design is both a complex adaptive system (CAS) and a component of other complex adaptive systems at the same time (Noun 3).

Practitioners from the fields of physical object and hard product design might argue that these differences are a matter of degree rather than a difference in kind. They could argue that because the field of software design is so young that appropriate first principles of design are not obvious or not yet discovered. To confuse the argument further, it is difficult to find an area of hard design that does not have a component of software vital to the design process. Software is often used for computer-aided design or simulation, as well as embedded into the product to make it more interactive. We believe the examples in this paper illustrate that there is a difference in kind with software design. Given our examples, and that software is a material without substance, we claim that it is likely that there are no first principles of software design to find.

We believe that in the future hard design processes will adopt and include many of the methods of successful software design practice. A good example of this melding of practices is documented in the shifting of design process and methods of the architect Frank Gehry. Through the last 20 years, Gehry's design practice has moved from that of the traditional architect developing drawings, specifications and models to adapting CAD and Computer Numerical Control (CNC) machining software to design and realize his complex building shapes. We discuss this below.

We also believe that in the reverse direction, some of the hard design practices aid in better software design, as demonstrated by John Socha in producing the highly successful product, Norton Commander. However, the differences between design in soft and hard systems require us to be careful as to which techniques to transfer from hard to soft, and vice versa. We elaborate on this below.

The rest of the article explores these four definitions of 'design' for software and how they are different or similar to design in other disciplines. The character of these four definitions is tightly coupled, so please bear with us as we explore each.

Design (Noun 1)—Concept of a field as a whole

Getting our minds around the field of design is difficult. Simon [2] states 'Everyone designs who devises courses of action aimed at changing existing situations into preferred ones.' Löwgren and Stolterman provide further context [3]:

> We live in an artificial world. It is a world made up of environments, systems, processes, and things that are imagined, formed, and produced by humans. All these things have been designed. Someone has to decide their function, form, and structure, as well as their ethical and aesthetical qualities. In this artificial world created by humans, information technology is increasingly becoming not only a common but a vital and fundamental part. Our designed world is full of *digital artifacts*, that is, designed things built around a core of information technology . . . To design artifacts is to design people's lives.

A wide range of engineering disciplines use a definition of design similar to that presented in Dym *et al*. [4]:

> *Engineering design* is a systematic, intelligent process in which designers generate, evaluate, and specify concepts for devices, systems, or processes whose form and function achieve clients' objectives or users' needs while satisfying a specified set of constraints.

The output of the engineering design process is generally a set of detailed specifications which can range from a simple text document to complex animated CAD drawings. Dym and Little [5] describe the scope and detail of the hard design process in steps that are recognizable for most design disciplines:

- client statement and need;
- problem definition;
- conceptual design;
- preliminary design;
- detailed design;
- design communication;
- final design (fabrication specifications and documentation).

Within the domain of software design, following the above steps in order is known as the waterfall method. When combined with powerful techniques and methods as described in Jones [6] and Cross [7], a designer from any discipline can build on the experience of others to produce good designs. An important development in the evolution of design methods is the shift to human centered design. Charles Owen [8] captures the scope and the tools of the Institute of Design's human centered design process in his lifelong work with Structured Planning. Nelson and Stolterman [9] describe a practice of design that is applicable to all fields. Schon [10] provides insights on how to improve design thinking through the process of reflection and double loop learning.

Dym and Little [5] point out that an important aspect of industrial design is that it is typically done with teams of individuals who not only include the product designers but also manufacturing and distribution engineers. Such teams design in parallel to optimize the product through its entire life cycle of design, building, distributing and operating. This process of concurrent engineering increases the complexity of the design process.

While we find these resources and points of view useful, none of them capture the unique difficulties of designing good software systems. We claim that the particular characteristics of the execution and operating environment of the end software design (Noun 3) make the process of designing software (Verb) different from design processes of most disciplines. In addition, we assert that the source code is the software design specification (Noun 2), since the manufacturing step has been essentially eliminated by widespread and powerful build tools that rapidly create the end product from the source code. Because the intent of the software product is to change how humans and human organizations act, the field of software design (Noun 1) needs to consciously cover the field of organizational development, as well as the field for which they are producing a product. Following Covey's 'Begin with the end in mind' adage [11], we cover the remaining definitions in the reverse order of Heskett's statement on design.

Design (Noun 3)—A finished product, concept made actual

What sets software design apart from other hard design disciplines is that the end product of the design is an interacting set of rules. Research in the science of complexity illustrates that even a few simple rules interacting with each other can produce complex behaviors. An exciting breakthrough in computing came from Chris Langton [12] at the Santa Fe Institute when he discovered that flocking behavior in birds could be simulated with three rules. Most of us assumed that there was always a leader of the flock, but there is not. Langton [13] showed that with only three rules he could emulate flocking:

1. Each individual shall steer toward the average position of its neighbors.
2. Each individual shall adjust its speed to match its neighbors.
3. Each individual shall endeavor not to bump into anything.

This insight and rule formulation started the field of generative computing. The World Wide Web has many examples of small programs which use these rules to emulate flocking behavior. One of the most popular applets looks at several levels of birds flocking with simulated birds called 'boids'. As you watch the animations, it is hard to realize that the complexity of behavior is coming from the interactions of three simple rules. Contrast this level of CAS with the average business program which has tens of thousands of rules interacting with each other.

Another example of a CAS with emergent behavior is the music that you listen to on the Sseyo [14] website. These are compositions done with a program called Koan. This program provides a visual interface to a CAS that generates music that is different with every playing. The interacting rules can be set at a number of different levels from manipulating the physics of sound to composing with visual icons of 'instruments' in order to generate compositions. Brian Eno, famous as a music producer, worked with the producers of Koan to refine their tool set. In his published diary, he describes the nature of generative music [15]:

> Ten RCA students over to look at Koan and screensavers. I gave them all a talk about self-generating systems and the end of the era of reproduction—imagining a time in the future when kids say to their grandparents, 'So you mean you actually listened to exactly the same thing over and over again.' Interesting loop: from unique live performances (30,000 BC to 1898) to repeatable recordings (1898-) and then back to—what? Living media? Live media? Live systems?
>
> Of course, the real can of worms opens up with the new stuff I'm doing—the self-generating stuff. What is the status of a piece of its output? Recently I sold a couple of pieces as film-music compositions (a minor triumph, and an indication of how convincing the material is becoming). I just set up some likely rules and let the thing run until it played a bit I thought sounded right! But of course the film-makers could also have done this—they could have bought my little floppy (for thus it will be) containing the 'seeds' for those pieces, and grown the plants themselves. Then, what would the relationship be between me and those pieces? There is, as far as I know, no copyright in the 'rules' by which something is made—which is what I specify in making these seed programs. The end of the era of reproduction.

Both the Boids and Koan programs are the results of a few rules interacting. Most programs used in business, games and consumer software products have tens of thousands of interacting rules. Further, these CAS software designs are embedded

in and surrounded by thousands of other software designs like operating systems, database systems, middleware and the World Wide Web infrastructure. The result is a living mixture of emergent behavior that seemingly mimics living systems. Even the vocabulary used to describe software systems—niche, virus, dead, adapting, evolving—is drawn from ecosystem literature.

We are beginning to have a better theoretical framework of what it means to be working with such CAS. Holland provides a high level view of CAS [16]:

> Overall, then, we will view CAS as systems composed of interacting agents described in terms of rules. These agents adapt by changing their rules as experience accumulates. In CAS, a major part of the environment of any given adaptive agent consists of other adaptive agents, so that a portion of any agent's efforts at adaptation is spent adapting to other agents. To understand CAS, we must understand these ever changing patterns.

Holland then identifies and describes the properties (aggregation, nonlinearity, flows, diversity) and mechanisms (tags, internal models, building blocks) that provide the foundation for a CAS theory. Potgeiter [17] builds on these CAS basics to describe how to design for emergent properties of CAS in software systems.

For software systems in particular, Highsmith looks more specifically at CAS, software design, and the management of the software development process [18]:

> In complex environments, adaptation is significantly more important than optimization. Adaptation includes the ability to utilize emergent order to alter actions that are essential if an organization is to survive and thrive in complex social and economic ecosystems. It includes the ability to make local alterations rather than depending on centralized, slow acting, control processes. Adaptation trades efficiency for speed and flexibility. Optimization works in a complicated world; adaptation works in a complex one . . .
>
> The greatest risk we face in software development is that of overestimating our own knowledge. . . . At the core of our ability to succeed in extreme environments is the admission that we don't know it all . . . Fast learning requires iteration—try, review, repeat.

What this means for software design is that software is never done, and that the behavior of the software on our computer can change without us asking for it to change. New behaviors emerge as the software is being developed, as it is put into operation, and as its environment (its ecosystem) changes. These quickly change both the user's and the developer's views of what is needed and what could be. Thus, software projects almost never deliver what they initially expected to deliver. Their plans are fluid in the extreme—not because people don't put in sufficient effort to plan, but because they cannot accurately predict the results of the CAS they are building.

Design (Noun 2)—A concept or proposal

As we look at the difference between what is espoused in formal courses and books on the role of software design specifications versus what happens in practice, we see that in practice the only software specification that meets the intent of the Dym *et al*. [4], design definition is the source code itself. As our tools for creating software have improved, it is much easier to get something started and get early user feedback with the real thing, than it is to design software in the abstract. Jack Reeves in an early article on designing software with modern languages describes this change in view [19]:

> The final goal of any engineering activity is some type of documentation. When a design effort is complete, the design documentation is turned over to the manufacturing team. This is a completely different group with completely different skills from the design team. If the design documents truly represent a complete design, the manufacturing team can proceed to build the product. In fact, they can proceed to build lots of the product, all without any further intervention of the designers. After reviewing the software development life cycle as I understood it, I concluded that the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code listings.

As the software engineering community created better development languages like Smalltalk, C++, Java and C# and robust development environments like Microsoft's Visual Studio, Eclipse, and IBM's Websphere, this assertion is becoming reality.

Reeves points out that another challenge of software design is that everything is part of the design process. There are no natural separations of functions as with hard engineering disciplines [19]:

> Coding is design, testing and debugging are part of design, and what we typically call software design is still part of design. Software may be cheap to build, but it is incredibly expensive to design. Software is so complex that there are plenty of different design aspects and their resulting design views. The problem is that all the different aspects interrelate.

As current development systems make it easy to start generating code (or is it design?), the software developer tends to jump in and start producing something without incorporating *any* explicit design methodology. Furthermore, the value of an up-front design is less compelling if CAS theory implies that we cannot understand what the design will produce until we execute the design.

An interesting outcome of Reeves definition is that once software is designed (written), it becomes very inexpensive to build and distribute. Reeves states [19]:

> There is one consequence of considering code as software design that completely overwhelms the others. It is so important and so obvious that it is a total blind spot for most software organizations. This is the fact that software is cheap to build. It does not qualify as inexpensive; it is so cheap it is almost free. If

> source code is a software design, then actually building software is done by compilers and linkers. We often refer to the process of compiling and linking as 'doing a build.' The capital investment in software construction equipment is low—all it really takes is a computer, an editor, a compiler and a linker. Once a build environment is available, the actually doing a software build just takes a little time. Compiling a 50,000 line C++ program may seem to take forever, but how long would it take to build a hardware system that had a design of the same complexity as 50,000 lines of C++.

Since Reeves' article was written, the performance of computing has increased exponentially, with the cost of building software approaching zero. Likewise, with the widespread adoption of the Internet and a shift to electronic downloads of software, the cost of distributing software is essentially zero.

These two changes mean that for software the distance between the concept or proposal (Noun 2) and the finished product (Noun 3) is approaching zero, which leads to changes in how we design software, as explained in the next section.

Design (Verb)—Indicating action or process

With the design of physical products, often the cost of making a realistic prototype is as expensive as the whole design process that goes before it. Therefore, the waterfall design method is often adopted as described in Dym and Little [5].

When designing software, on the other hand, Dorst describes a different management process [20]:

> If you look at web design, for instance, you would see quite a different pattern. In developing a website or an interactive system for a computer, you work on designs that are easy to replicate, and that will be used by means of the same medium on which they are made. So you have a realistic 'prototype' at almost any moment during the design process. You can do user testing at all times. Designing then changes from a linear process which leads to a prototype, into a process of continuous testing and learning. Design becomes an evolutionary process; you are able to test many generations of the design before delivery.
>
> Evolutionary development is wonderful: the earlier you can incorporate user knowledge into the design, the better. Unfortunately, in practice it turns out that these evolutionary processes are even harder to manage than 'normal' design projects. How do you decide on the number of generations you will need, for instance? This way of working also has its own pathology, the results of which are all too familiar: the debugging drama. Software designers are tempted to 'just make something' and then to improve that imperfect concept over many generations. But if you begin the evolutionary process at a level which is too detailed, you end up debugging a structurally bad design, ultimately creating a weak and unstable monster.

The evolutionary design process described by Dorst also has another challenge: getting the right level of feedback from the client and the user. This contrasts with hard design where significant effort is expended in making a realistic prototype. Because software designs look so usable at an early stage, the users want to jump right into using the design and the result is feedback that is at the myopic level, not at the reflective and systemic level.

A technique for getting better feedback at this early level is to change the resolution or fidelity of the design. Paul Souza, while at Adobe Corporation, developed a technique of 'animating' pencil sketches. Instead of a polished user interface with a set of actions and data models developed underneath, he would scan a pencil drawing into the computer and assign hot spots to the drawing in order to call a function. With a 'polished' user interface the only kind of feedback he would get would be on the font and the colors and layout of the interface (convergent detailed feedback). With the pencil sketch interface on top of the actions and data model, he would get conceptual feedback about the intent of the tool and how the tool might be used to better the organization's goals (divergent and generative feedback). Also, by lowering the fidelity of the user interface, he reduced the demand to prematurely start using the design before a robust architecture could be formulated.

A modern development technique, called Test Driven Development (TDD) [21], illustrates how the malleability of software can change lower level designing. In TDD the classic sequence of engineering steps – *What to do?*, *How to do it?, Do it!* and *Did we do it?*—are inverted. First we write a test (hence the name) that asks the *Did we do it?* question. The initial answer is usually 'No.' We then *Do it!* by coding until the test passes. We then consider the existing design and make improvements—the *How to do it?* question. Typically, we then more fully explore the *What to do?* question which often leads us to the next *Did we do it?* question. TDD is part of the Agile methods of development that include eXtreme Programming and Scrum.

Designing using TDD has some interesting characteristics. The designs themselves are often simpler than more traditionally designed solutions. The process spins off tests which add considerable value as they continue to be used during development to enable reliable change. The long-term feel of the design process is considerably different. Some folks characterize it by saying that designing is so important they do it all the time rather than just 'in the beginning'. The feel of the work is different. It has more of a character of a dance since the steps are repeated frequently. Better software designers tend to take smaller steps.

A weakness in the approach is that once the *Do it!* step has been completed, there is strong pressure to not accomplish the next step, *How to do it?*, the design step. After all, the code works, why change it? If it is not done, the design of the software rather quickly degenerates and becomes brittle and hard. This is the character of software developed using classic techniques, and it points to the hardening of software as primarily a design issue.

All of these differences change the way that we need to manage the process of designing software. Managing any non-trivial software project requires techniques that honor and take advantage of how complex adaptive systems work and the emergent behavior they generate, for not only is the software developed by a CAS (the software development organization) for a CAS (the user community) the actual product is a CAS whose behavior cannot be predicted. In such situations, an empirical process control mechanism that creates an environment providing frequent and regular feedback on what has been built, such as those employed in eXtreme Programming and Scrum, work better than the traditional waterfall process control mechanisms.

Design (Noun 1)—Concept of a field as a whole

The preceding sections describe the differences during the process of designing, building and distributing software products but not how the product is used. Most interesting software is used in an organizational context, which creates another challenge.

Successful software design processes include an additional stage of design activities: organizational design and intervention (Intervene). While any good designer must span knowledge domains such as the problem domain and the solution domain [9], the nature of software design causes the designer to span more knowledge domains than other designers. A good designer in any field will understand the design brief from the purchaser and then do research on the users' needs. Yet, most interesting software is used in an organizational context. One could argue that the modern corporation is only as good as the software that it employs. Much software is, after all, automating things that people could, or did, do before without software, or extending what they could, or did, do before. So using the software will require people to change what they have been doing. Thus, for the software to be effective and usable at its introduction, the software designer needs to understand the basics of organizational development and realize that software development is an organizational intervention.

Floyd identifies this key attribute [22]:

> Enterprise information systems codify structural aspects of organizations. They come with problems of integration and (organizational) standardization on a large scale. Usually it is not a question of developing new systems but of adapting existing systems, so design pertains to how to introduce the system in the organization at hand. Technical challenges lie in using components for tailoring systems to specific needs. The relevant social context is organizational development. Software practitioners are engaged in organizational intervention, being perceived as agents of change. They also have the role of mediators between organizations and vendors.

While Floyd only made the organizational intervention argument for the scale of enterprise information systems, we assert that most software design is an organizational intervention. However, most software developers do not take organizational design into consideration explicitly.

Löwgren and Stolterman relate the challenge of organizational interventions with the personal, social and political aspects of designing digital artifacts [3]:

> If a design process aims to create an information system in an organization, then individuals, groups, and teams can be seen as kinds of material. The challenge is to design the social 'components' together with the technical components as a systemic whole . . . Designers of digital artifacts face a particular difficulty. The material they use—that is, the digital technology—can in many ways be described as a *material without qualities* . . . As a consequence, the design process becomes more open, with more degrees of freedom and therefore more complex.

> Design is also a political and ideological activity. Since every design affects our possibilities for actions and our way of being in the world, it becomes a political and ideological action. With designed artifacts, processes, systems and structures we decide our relations with each other, society, and nature. Each design is carrying a set of basic assumptions about what it means to be human, to live in a society, to work, and to play. When looking at large infrastructural designs, such as the way we organize society and companies or large technical systems, most people realize how they affect the way we can live our lives. We would like to point out that the same also holds true in a small-scale perspective. Every digital artifact restricts our space of possible actions by permitting certain actions, promoting certain skills, and focusing on certain outcomes. To some extent, the user has to adapt to the artifact . . . The role of digital artifacts has to be recognized and measured in relation to the way they have a real impact on our lives.

Further, the software designer needs to understand the impact of the software design on at least three levels of organizations—the using organization, the customer of the using organization, and the software development organization itself. Since most software projects of significance last from one to five years, the software designer must look at the today state (As Is) of each organization and make a projection for what the future state (To Be) of each organization is likely to become. Software designers need to be schooled in the basics of organizational development, as well as the aspects of team development of 'forming, storming, norming, performing, and adjourning' [5].

Over the course of my career, I (Skip) alternated between line management jobs in software engineering and working as an organizational consultant helping large and small organizations develop visions, missions, strategies and innovative product designs. In the process of consulting and graduate school teaching, I tried to pass on what I've learned about designing successful software products and systems. While my customers and students generated better designs, they did not generate innovative designs like I've accomplished

over my career. I knew there was something missing from my framework of design, but I couldn't pinpoint it.

Then I had a Chris Alexander [23] moment while reading Floyd's article. Alexander realized that the reason his students weren't producing great designs is that he left two important aspects out of his Pattern Language—color and asymmetry. Similarly, I left out of my teaching the foundations of organizational development, change and design. Yet at least half of the work of every successful product design that I've done has included innovative organizational design and interventions.

Having even simple models of organizational and process design improves the quality of the design process and the resulting designs. There are many such models. Ackoff [24] with his idealized design provides both a simple and a robust methodology for charting an organization's future. Fritz [25] with his structural tension model provides both a personal and an organizational model for development. Rummler and Brache [26] provide an organizational view of the process flows in an organization that reminds us that an organization chart is not the only means of viewing how an organization works. Goldratt [27] provides a view derived from the types of thinking in physics on how an organization can change through the focusing on the constraints inherent in any organization or work flow.

DESIGN—HARD AND SOFT

In my (Skip's) graduate course on 'Creating Products Interactively', I assert that all product development is essentially a software and information design problem today. The assertion stems from the increasing use of specialized CAD tools to aid in the design of products, along with more computing being embedded in hard products. Yet, most corporations that produce hard products still deal with software as an afterthought. At a recent Center for the Advancement of Engineering Education (CAEE) [28] review meeting, the Director of the Design Institute for Global Core Engineering from a major automobile manufacturer described the major curriculum subjects of the institute. They were all about the physical components of the car. When asked where was software design within the curriculum, he replied that it was subservient to the major functions like Powertrain and Control. He then indicated that this could be a problem in the future as more functions are moved from mechanical designs to computing and software designs. I (Skip) own a Mini Cooper car and have had three recall notices—all to fix software problems.

Similarly, Adidas recently released the Adidas-1 computerized running shoe for continuously adapting the shoe to the demands of the runner [29]. The brain of the shoe is located under the arch and is capable of making 5 million calculations per second and 1000 readings per second from the sensors to the shoe's computer. The software and sensors judge whether the cushioning is too soft or too firm and adjusts the fit throughout the run. Is it a shoe or a computer? Is it hard or soft design? Clearly, the answer is both. This shoe is an example of the wide range of domains that the design team had to cross to produce a viable, interactive physical product.

In a more complex example, Frank Gehry, at a Technology, Education and Design (TED) Conference put on by Richard Saul Wurman, described his challenges in creating the kind of public building designs such as the Guggenheim Museum in Bilbao, Spain, the Experience Music Project in Seattle, and the Disney Concert Hall in Los Angeles. When he first started exploring complex curved shapes for the exterior of buildings he was startled to discover that when he put his designs out to construction bid, the contractors quoted him five times the normal fees. He realized that no one knew how to build his creations. So he had to form a company to first adapt CAD tools to design the complex metal shapes, and then develop the software that would connect his CAD tools with CNC equipment to cut and mill the complex metal shapes. The end result was that he was able to build his distinctive creations for the same cost as traditional construction methods. During his presentation he reflected on whether he was now a building architect or a software designer.

These changes are causing the field of architecture to look more like the field of software design. Lindsey details the extent to which computer systems and particularly the Dassault CATIA CAD system [30] have entered Gehry's practice of architecture. The computer is used for simulations of the digital and physical models, direct detailing, computer aided manufacturing, coordination of the electrical, mechanical and plumbing systems, and as a framework for the operation of the building after construction. Gehry describes how his evolving process is changing the craft of building design and construction [30]:

> This technology provides a way for me to get closer to the craft. In the past, there were many layers between my rough sketch and the final building, and the feeling of the design could get lost before it reached the craftsman. It feels like I've been speaking a foreign language, and now, all of a sudden, the craftsman understands me. In this case, the computer is not dehumanizing; it's an interpreter.

The significance of the changes that Gehry has made in his fluent design process shows up in the organizational interventions that the software is bringing to the building industry [31]:

> Ultimately, allowing for all communications to involve only digital information, the model could signal a significant reduction in drawing sets, shop drawings, and specifications. This is already reflected in the office's current practices where the CATIA model generally takes precedence (legal as well as in practice) over the construction document set. This is a

> significant change in standard practice where specifications take precedence over drawings and specified dimensions are subject to site verification. . . . . Glymph states that 'both time and money can be eliminated from the construction process by shifting the design responsibility forward'. Along with this responsibility comes increased liability. When the architect supplies a model that is shared, and becomes the single source of information, the distributed liability of current architectural practice is changed.

Building on the experience of Gehry, we see that this combined hard and soft design can shift forward into the area of operating a building as well. One software system can act as a shared repository and information refinery for the design, build, distribute, intervene and, now, the operate phase knowledge base.

Likewise, the software design discipline can learn from the processes of hard product design. Some of the best software designers come from physical science or engineering disciplines where they learn early on the power of constraints. Having a background in designing with physical parts appears to provide a different perspective on design, in part because you learn to think about and design systems that respect the hard constraints of the physical world. John Socha, author of the Norton Commander software package, attributes much of his software design success to his electrical engineering background. Unlike computer scientists, electrical engineers spend a lot of time dealing with failure modes, since they cannot count on clean signals coming into their parts. He applied this to software design *not* by creating rigid software that enforces interface standards, but by creating software that does the most reasonable thing when its inputs are out of the expected range. The result is software that fails gracefully.

THE SPAN OF DESIGN

As we look at the differences between hard and soft design along with their merging, the span of the domains that need to be designed for emerges. Good hard and soft design in the future needs to encompass these stages:

- Design
- Build
- Distribute
- Intervene
- Operate

The amount of time and resources required for each of these stages varies considerably between hard and soft design.

For hard design, the resource expenditure looks like:

Design **Build** Distribute Intervene Operate

For soft design (e.g., software), we see very different resource utilization:

Design Build Distribute **Intervene** Operate

The magnitude of the above resources shows how the focus of the designer must shift depending on what they are designing. We believe that this graphic provides guidance for what, and how, we should be teaching design across the hard and soft disciplines.

DESIGN—TEACHING

The common understanding of what and how to teach software design is at a very low level. In September 2004, as I (David) was coming out of the main office of the Allen Center at the University of Washington, I saw Ed Lazowska waiting for the elevator. Ed is an impassioned and impressive researcher, teacher, and politician with wide contacts inside and outside the university. I've always respected his opinions and his astute observations, so I took the opportunity to ask him some questions.

'Do you know,' I asked, 'whether the professors here believe that computer science is a mathematically-centric discipline, or a design-centric discipline?' He thought for a moment, and then replied that he believes most of the professors in this building believe computer science is a design-centric discipline. They practice design as algorithm design, system design, etc. However, because they have no formal training in design, they don't know how to teach design. Which may be why there are so few courses on software design.

Instead of coming from a discipline, like Civil Engineering, where students are introduced to the concepts of design at an early stage, the professors in computer science have never been formally taught about design, and thus don't know how to teach it to their students. This is the same with how to do research—they are excellent researchers, but they have no training in how to teach it.

Even the current focus on software design patterns [32] is at a low level, aimed at establishing a design language [33] to describe design (Noun 2). It is creating a common vocabulary. However, we see very little work on the verbs and the grammar rules for good composition.

Given this setting, how can we bring to the software discipline a coherent and effective model of software design that fits the forces at work in software?

We hypothesize, based on recent experience in teaching senior level software development courses and an industrial design course, that the emergent trend toward applying agile development techniques in the classroom will lead to a better appreciation for the issues covered in this paper. We are aware of courses in human centered design (Institute of Design [34]), Human Computer Interaction Design (Indiana University [35]), and Personal Fabrication (Gershenfield MIT [36]) that are achieving success helping students produce designs

using the integrated methods of hard and soft design described previously. While many of the areas of software designing can be improved through this process, a key area of concern is how to teach the organizational intervention component of software designing. Where can students experiment with and learn about organizational development, since most organizations are reluctant to let experienced professionals loose in their organizations, let alone student practitioners?

We also hypothesize that the entire undergraduate experience would be substantially enhanced if design language and process were introduced in the very beginning of the undergraduate education and then referred back to, enhanced, etc. in every course. After all, virtually every course is about design, whether it is the uncovering of the designs that exist in nature, understanding the designs created by others, or creating new designs. This type of curriculum intervention has a low probability of being accepted by most higher education organizations, because it reframes the concept of a curriculum to have coherent threads passing through the entire curriculum. As a result, industry is left to use apprenticeship to try and impart more appropriate design practice.

SUMMARY

Over the last ten years, we have seen improvement in the field of software designing by drawing on the knowledge from other design fields. The adoption of pattern language techniques from the field of architecture provides consistent solutions to low-level software design tasks that arise repeatedly. Designing from human-centered techniques versus technology-centered techniques has sped up customer adoption of new products. Brainstorming methods, team process methods, and science of design methods have all helped produce better software more productively. While these processes form firm foundations for other disciplines, and are valuable in software, they are not sufficient for designing software.

While Heskett's 'Design is to design a design to produce a design' seems nonsensical at first reading, it serves as a clarifying framework to look at the similarities and differences between hard and soft design. The view of the design activities field needs to expand beyond just generating a specification to include the full range of activities—design, build, distribute, intervene, and operate. The evolution of Frank Gehry's and John Socha's respective design experiences suggest that the future of hard and soft design is not an either/or choice but rather the appropriate combination of techniques, skills, and processes.

As we see in this paper, each of the meanings of 'design' is different between software design and most other design disciplines. In software, the first noun extends to include organizational intervention as a significant component. The verb is about reversing the steps (test-driven development). The second noun is about the source code is the design specification. The third noun is about the design not being an object to be manufactured, for most classes of software, but instead being a complex adaptive system that is an organizational intervention. The other design disciplines that are most like software design are those that share these qualities, including the disciplines dealing with bioengineering and social systems.

The result is what we believe to be a convincing story that software designing is different because it is a field of a 'material without qualities'. The key differences are:

- Source code is the design (Noun 2)
- Design (Noun 2) and organization intervention (Noun 1) are the dominant steps, unlike hard design where build and distribute are the dominant steps
- The steps of the waterfall design (Verb) model are reversed
- There is little material resistance with software—no physics, no first principles, no simulation from first principles
- Software design (Noun 3) is a complex adaptive system design
- Software is always deeply embedded—it exists in some hardware form which provides one set of constraints, and in a soup of other complex adaptive systems which generates fuzzier constraints

As software becomes more prevalent in the design tools and products of other disciplines, we can expect those 'hard' disciplines to become progressively more 'soft', with the concurrent change in forces requiring softer design techniques. In the future, many domains of hard design will require multiple design methods and processes:

- Hard design processes for those things that physics apply to.
- Software design processes for the software or 'alive' components.
- Recognition that most products have an organizational intervention component to them.

While we identify key differences between software design and hard design, the awareness of these differences is little understood by practicing software designers. As we write this article, there are further challenges coming on the horizon that will affect how software designers work:

- Availability of cheap multi-processor personal computers with the introduction of multi-core processors—four processors per chip this year expanding to sixteen per chip in the near term [37]. How will we effectively harness this parallelism to continue to deliver improved software performance?
- Expansion of media types used on a daily basis

in business from text and numbers today to sound, pictures, and personal fabrication of interactive physical objects in the near term. [38] How will education and business be transformed by the regular use of multiple media?

Expanding software designer capabilities to include differences identified in this paper, along with the technology and business needs we see coming, will strain university curricula and all forms of knowledge acquisition and transfer. We expect this change to take years—after all, it has taken professional software developers twenty years to widely accept object-oriented technologies. We look forward to contributing to a unifying and pragmatic model of software design, better software design tools, and engaging student and professional curricula to improve the field of software design.

Acknowledgements—The authors thank Jeff McKenna, Steve Forgey, Barney Barnett, Wolf-Gideon Bleek, and Robin Adams for their thoughtful comments and insights into the software design process and design education which contributed to this article. This research was supported in part by NSF Grant number EIA-0121326, and by the Institute for Scholarship on Engineering Education through the NSF funded Center for the Advancement of Engineering Education (NSF ESI-0227558).

REFERENCES

1. John Heskett, *Toothpicks & Logos: Design in Everyday Life,* Oxford University Press, Oxford (2002). Portions of the presentation can be found at http://www.johnheskett.net/page01.htm
2. Herbert A. Simon, *The Sciences of the Artificial*, The MIT Press, Cambridge, MA (1969).
3. Jonas Löwgren and Erik Stolterman, *Thoughtful Interaction Design: A Design Perspective on Information Technology*, The MIT Press, Cambridge, MA (2004).
4. Clive Dym, Alice Agogino, Ozgur Eris, Daniel Frey and Larry Leifer, Engineering design thinking, teaching, and learning, *J. Eng. Educ.,* January 2005.
5. Clive Dym and Patrick Little, *Engineering Design: A Project Based Introduction*, John Wiley and Sons, Hoboken, NJ (2004).
6. John Chris Jones, *Design Methods*, John Wiley and Sons, New York (1992).
7. Nigel Cross, *Engineering Design Methods: Strategies for Product Design,* John Wiley and Sons, Chichester, UK (2000); Nigel Cross *et al*., *Analysing Design Activity*, John Wiley and Sons, Chichester, UK (1996).
8. Charles Owen. http://id.iit.edu/papers/Owen_theoryjust.pdf
9. Harold Nelson and Erik Stolterman, *The Design Way: Intentional Change in an Unpredictable World: Foundations and Fundamentals of Design Competence,* Educational Publishers, Inc, Englewood Cliffs, NJ (2002).
10. Donald Schon, *The Reflective Practitioner*, Basic Books, New York, 1983.
11. Stephen R. Covey, *The 7 Habits of Highly Effective People*, Simon and Schuster, New York (1989).
12. John Horgan, From complexity to perplexity, *Scientific American*, June 1995. http://www.econ.iastate.edu/tesfatsi/hogan.complexperplex.htm
13. http://www.vergenet.net/~conrad/boids/
14. http://www.sseyo.com
15. Brian Eno, *A Year with Swollen Appendices*, Faber and Faber, London (1996).
16. John H. Holland, *Hidden Order: How adaptation builds complexity*, Helix Books, Reading, MA (1995).
17. Anna Potgieter, The Engineering of Emergence in Complex Adaptive Systems, Ph.D. thesis, University of Pretoria, 2004. http://upetd.up.ac.za/thesis/available/etd-09222004-091805/
18. James A. Highsmith, III, *Adaptive Software Development: A collaborative approach to managing complex systems*, Dorset House Publishing, New York (2000).
19. Jack Reeves, What is software design? *C++ Journal*, Fall 1992.
20. Kees Dorst, *Understanding Design: 150 Reflections on Being a Designer,* BIS Publishers, Holland (2003).
21. Kent Beck, *Test Driven Development: By Example,* Addison-Wesley, Reading, MA (2002).
22. Christiane Floyd, Developing and embedding auto-operational form, in Dittrich *et al*., *Social Thinking—Software Practice*, MIT Press, Cambridge, MA (2002).
23. Christopher Alexander's extensive body of work includes *Notes on the Synthesis of Form, Timeless Way of Building, Pattern Language*, and his four volume series on *The Nature of Order.*
24. Rusell Ackoff, *Creating the Corporate Future: Plan or Be Planned For*, Wiley, New York (1981).
25. Robert Fritz, *Path of Least Resistance: Learning to Become the Creative Force in Your Own Life,* Ballantine Books, Boston (1989).
26. Geary Rummler and Alan Brache, *Improving Performance: How to Manage the White Space in the Organization Chart*, Jossey-Bass, Boston (1995).
27. Eli Goldratt, *The Goal*, North River Press, Hartford, Connecticut (2004).
28. http://www.engr.washington.edu/caee/
29. The Daily Reveille, Adidas introduces computerized running shoes, http://www.lsureveille.com/vnews/display.v/ART/2005/03/17/42392654eebf1
30. Dassault CATIA web site. http://www.dassault.fr/en/valeur.php?docid=156
31. Bruce Lindsey, *Digital Gehry: Material Resistance, Digital Construction*, Birkhauser, Basel (2001).
32. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, New York (1995).
33. John Rheinfrank and Shelley Evenson, Design Languages, in *Bringing Design to Software*, edited by Terry Winograd, Addison-Wesley, Reading, MA (1996).

34. Institute of Design website. http://id.iit.edu/grad/welcome.html
35. Eli Blevis *et al*., Integrating HCI and Design: A Design Education Case Story. http://www.informatics.indiana.edu/eblevis/designandhci.pdf
36. Neil Gershenfeld, *FAB: The Coming Revolution on your Desktop—From Personal Computers to Personal Fabricators*, Basic Books, New York (2005).
37. Herb Sutter, The free lunch is over: a fundamental turn toward concurrency in software, *Dr. Dobb's Journal,* **30**(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm
38. Stan Davis and David McIntosh, *The Art of Business: Make All Your Work a Work of Art*, Berrett-Koehler Publishers, San Francisco (2005).

David Socha studies the human side of software development. He currently is the Software Project Manager on the UrbanSim project, and a Lecturer in the Computer Science & Engineering department, both at the University of Washington, Seattle where he received his Ph.D. in 1991. After his doctorate, he spent 11 years in industry, 6 of those managing teams of software developers, before returning to practice software development in academia.

Skip Walter is CTO of Attenex Corporation bringing over 35 years of technology product development experience along with executive management experience in Fortune 1000 companies and start-up businesses. Skip was the creator of DEC's ALL-IN-1, a $1 billion revenue per year office automation system. He taught Masters and PhD courses in interactive product planning and tangible knowledge design at the Institute of Design at the Illinois Institute of Technology for 10 years.