Examining Software Design from a General Design Perspective

Alex Baker and André van der Hoek
Institute for Software Research & Department of Informatics
University of California, Irvine
Irvine, CA 92697-3440 USA
{abaker,andre}@ics.uci.edu

ISR Technical Report # UCI-ISR-06-15
October 2006

Abstract

Our community's understanding of software design has certainly improved over the years, as have our approaches and our ability to design high-quality software. But compared to other design disciplines, the software design field is unable to objectively answer some fundamental questions about where it stands. Through a new, interdisciplinary framework, we place software design in a general design perspective, examine existing conceptual points of view and technical contributions, and suggest promising research directions. In so doing, we learn that we can and must treat software design as an inherent design discipline, that the field's successes to date have been in line with such a view, and that the framework can serve as a basis for engaging in the discussion necessary to answer the questions that have remained elusive to date.

1. Introduction

Design has long been recognized as having a critical role in software engineering. With the ever-increasing complexity of the software systems we develop, this role has certainly not diminished. A high-quality design can make the difference between a software system that is successfully developed, deployed, and used, and one that fails utterly somewhere along the way.

Given this critical role, it is no surprise that the software engineering literature contains a multitude of proposals and opinions regarding design approaches, modeling notations, evaluation techniques, tools, and other innovations. But, to date, these contributions have been discussed in a disjointed, piecemeal fashion. Notations have been debated in terms of expressiveness and verifiability, analysis algorithms in terms of properties that are guaranteed and algorithmic complexity, and design environments in terms of feature lists. Such criteria are vital to advancing each of these research topics, but we observe that the field as a whole lacks the ability to put all these topics and contributions in a shared understanding of software design.

What are the critical factors at play in software design? How do these factors influence our approaches? Where does our field excel and where do our efforts fall short? What should our basic research goals be? Answering these questions is difficult. Part of the problem is that when one considers software design in terms of a particular process model, project size, or language, this drastically shapes the concept of software design one is likely to have.

We wish to describe the fundamental nature of software design unburdened by the assumptions of individual research contributions. We do not want to assume that software design is a phase, or in the code, or to be expressed formally. Rather, we wish to tackle software design's essential nature. To do so, we have taken the drastic step of removing ourselves from the concept of software entirely, and have built an interdisciplinary framework that defines design in terms of factors that underlie *all* design disciplines. We recognize that software is different from teacups, cars, and billboards, but that does not mean the field can just ignore the fundamental nature of design. Rather, it must explicitly embrace it, understand how software represents a unique challenge, and work towards addressing this challenge from a design perspective.

In this paper, we derive our framework and use it to explain the challenges inherent to designing software. This understanding is then used to evaluate our field's ability to meet the needs of a designer, in terms of its current languages, tools, and shared knowledge. Based

on this evaluation, we make several suggestions of new research directions for improving software design.

The key lesson we learn is that we can and must treat software design as an inherent design discipline. Doing so allows us to explain the field's contributions in terms of core design concerns, lay out an agenda for future research, and use our framework as a basis for the discussion necessary to answer the aforementioned vital questions on which agreement has been elusive.

2. Our Framework

Our framework describes design as an information-based process, but in order to explain design in this way, we must first adopt a concept of the design product itself. In this section, we discuss the design product and process, and then present a brief discussion of how they can be used to explain communication among teams, stakeholders, and design communities.

2.1 Design – The Product

A design product is typically envisioned as a tangible artifact that is the result of a design process: a blueprint, sketch, mock-up, or any other form of expression that shows the intent of the designer. This, generally, is the view in software as well, with the design document being the predominant example.

General design theory has instead adopted more abstract models that resolve around the notion of a *design space* [1]. Each state in this space represents one particular design, with certain properties distinguishing it from other states representing other designs. A design may be complete or incomplete, and can be tangible or purely mental. A design process is described in terms of the designer adjusting the design, and thereby moving from state to state and exploring the design space. Of course, the notion of a design space is entirely hypothetical, as it would be impossible to ever articulate all of the designs that would theoretically populate it.

But what do individual states capture? We consider a given design to be an abstraction that constrains a set of possible outcomes. A vague design describing a few features allows for countless possible realizations of those features. But as a design becomes more precise, some of these realizations are excluded, until a relatively small subset of outcomes are described.

The goal of the designer is to create a design that describes only outcomes that are *feasible* and *desirable*. Certain designs may not be feasible due to limitations of the materials they describe (i.e., we can design, but not build, a skyscraper resting atop a narrow pole). In addition, the design must describe a desirable outcome that adds value in the eye of its beholders (i.e., we only want skyscrapers that can survive the effects of earthquakes and wind). Desirability stems from customers, who set up goals for what they want, as well as designers, who typically have their own sense of appreciation for certain outcomes.

Figure 1 illuminates this discussion. In the space of all conceivable outcomes (C, containing every outcome we could ever imagine, whether it can be realized or not), a designer has a notion of which are generally feasible (F) and desirable (D) outcomes, the latter being rooted in their experience and broad knowledge of what the field considers "good" and "bad". In addition, the customer has a concept of what is desirable (D) for this project. Working in the hypothetical design space, the designer converges on a particular design state (☆), which maps onto a set of still-possible outcomes (SP) that all fit the given abstraction. A *successful* design, then, is a state in the design space that describes only outcomes that are feasible, and that both the customer and the designer consider desirable.

It is important to note that a design rarely describes a single outcome; even a very precise design usually allows for some variation in the ways that it might be realized. But as long as each resulting outcome remains within the boundaries of feasibility and desirability, the design is sufficient. When a design allows for interpretations that would prove impossible midway through the implementation process, or that would prove distasteful upon completion, the design must be refined to exclude such outcomes, if it is to be considered "successful". Of course, predicting whether this is the case or not can be very difficult.

This concept of abstractions of outcomes also applies to the goals that a customer sets as input to the design project. Although a specific outcome is usually envisioned, a goal statement generally is imprecise and describes not a single outcome, but criteria describing the set of desirable outcomes. Through these examples, we can see the importance of describing our outcomes in terms of sets. Because design is an information-based process, designers are not employed to create concrete realizations, but rather abstractions, descrip-

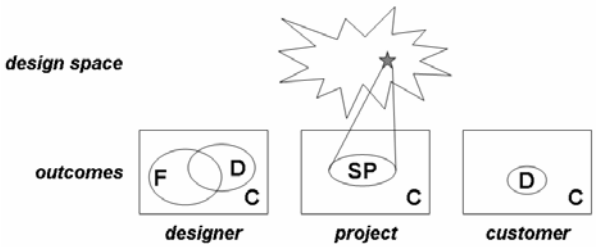

Figure 1. Design Space and Outcome Sets.

tions of them that will ensure their success.

As a designer moves through the design process, the set of still possible outcomes changes constantly. While much of the design process is about finding the choices that cause a design to fit inside the sets of desirable and feasible outcomes, these latter spaces are not necessarily static themselves. The invention of new materials, for instance, broadens the set of feasible outcomes. As another example, a goal that is impossible to attain may need to be broadened through discussion with the customer. A designer, thus, has several avenues available when trying to find a successful design.

2.2 Design – The Process

A design process is usually considered some kind of phase in a larger endeavor. A canonical example is the architecture design process as the precursor to the construction of a new building, but we can also consider the artist who mentally organizes a plan for their next sculpture or a fashion designer sketching a new tuxedo.

As with our treatment of the design product, we find a need to step back from the notion of the design process as a phase. Instead, siding with general design theory, we choose to begin discussion of the design process as a mental process. Faced with the task of finding a successful design, what is it that actually takes place inside the head of a single designer?

Naturally, the answer to this question involves some kind of exploration of the design space, as discussed by Newell and Simon [1] or Smithers and Troxell [2]. This exploration is characterized as one of information manipulation, with three different kinds of information involved: knowledge, goals, and ideas [3]. *Knowledge* is preexisting information that a designer brings to the table when faced with a design problem, including their experiences, judgment, results, and insights. In some cases, knowledge is factual, revolving around specific rules for achieving feasible and desirable outcomes under certain conditions. But knowledge can also be very intuitive, as with the designer's sense of aesthetics: their feelings about what kinds of outcomes are desirable and how to achieve them. But the important distinction is that a designer's knowledge exists independent from the design projects they undertake.

The second kind of information is the *goal*, which frames and guides the problem to be addressed. A goal is generally unique for a given design project; if it were not, an existing solution could be reused, and design itself would be unnecessary. A goal directly connects to the notion of desirability and may shift in response to fluctuations in a customer's preferences and priorities.

The form of the design itself is captured by the third form of information: *ideas*. Ideas are specific notions that together define one or more states in the design space. Ideas can take all sorts of forms, including vague intuitions, firm decisions, relations among ideas, thoughts on possible directions, preferences, and rationale. The general process via which ideas coalesce into a single, coherent design has been documented as one of divergence, transformation, and convergence [4]. When ideas are generated, organized, and used, they can be considered cuts across the design space, criteria that include some outcomes and exclude others.

But mental exercises alone are insufficient for a typical design process. Simon recognizes designers' needs to externalize their thoughts, and discusses the ways that this can aid mental processes [3]. Schön observes the phenomenon of a reflective conversation with materials, which he informally describes as an externalized form of a design "talking back" at its designer, giving insight into the outcomes it describes [5]. And, unsurprisingly, communication is an integral part of the design process, whether it pertains to absorption of the initial goal, learning from the literature, or discussing a design problem with others. Any such interaction uses a *representation*, which is defined as a concrete expression of mental forms of information. Such expressions often will concern ideas, but they may equally describe knowledge or goals.

The design process, thus, consists of a constant interplay among these four types of information to create, modify, and purge information, as needed. This interplay transpires through individual design *activities*, which a designer undertakes to find a successful design in the design space. Activities can be purely mental, as when a designer remembers, weighs alternatives, or makes a personal decision, but can also involve tangible aspects, as when the designer creates a sketch, absorbs the experience of sitting in a clay mock-up of a car, or performs research by looking at buildings that have a purpose similar to the one to be designed.

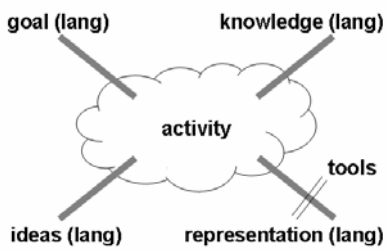

Figure 2. Design as Information Manipulation.

The view of activities manipulating the four kinds of information constitutes the basis of our framework as it pertains to the design process (see Figure 2). Countless activities are possible, but these can be categorized according to the interactions between the four information types. For instance, certain activities build up new knowledge (e.g., generalizing ideas, reading literature), others put this knowledge to use (e.g., conceiving ideas, evaluating alternatives), and yet others adjust the goal (e.g., reading a customer memo, building a scale model to see if the planned design shape is desirable).

All information, regardless of whether it is knowledge, goal, idea, or representation, is stored in some kind of *language*. The reader of this paper is likely familiar with English, and design disciplines frequently use specialized languages, such as graphic designers' use of Pantone codes to describe colors, or software designers' use of UML. Languages need not be symbolic either, and may rely on visual or gesture-based conventions.

Not all information is stored in the same language. On the contrary, ideas take on many forms and employ many ways of being stored (e.g., one might express the idea "I want to use red and blue in this painting" verbally, while using a sketch to plan the painting's layout). A designer's knowledge and the project's goals are similarly stored in different languages, including, among others, text, images, charts, and symbols.

The final part of our framework is *tools*, which aid designers in their interactions with representations. This concept includes familiar tools such as scalpels for manipulating clay models or drawing tables for creating blueprints. With the advent of computers, however, software has taken on a significant role in many disciplines. Software tools now not only help in creating representations (e.g., CAD/CAM diagrams, graphic designs), but also in interpreting them (e.g., automated analyses, search features or image enhancement tools).

2.3 Team, Stakeholders, and Community

Thus far, our framework has concerned a lone designer; but design does not take place in isolation. A designer may be part of a broader design team, involving multiple stakeholders (forming a community of interest [6]), and a designer is generally part of one or more design communities (forming communities of practice). Each of these provides a context in which a designer must communicate with others.

We consider any information exchange among multiple people as *communication through a shared representation*. That is, for two or more people to communicate some knowledge, ideas, or goals, it is necessary for such mental information to be expressed in a tangible form. Such a representation may be an elaborate document, but may also involve sketches, formal presentations, spoken words, scribbles, or gestures.

Our framework sheds light on several difficulties inherent to communication. First, few representations are complete and accurate with respect to the mental thoughts they are meant to express. In general, a representation is an abstraction, subject to (re)interpretation by its recipient. It is the hope of the designer that a recipient interprets the material in a way that is close enough to their intention [7]. This depends to a large degree on whether those communicating share common ground, allowing them to fill the gaps in their communication with shared assumptions [8]. If goals, ideas, and knowledge are similar enough, low-precision representations might be sufficient. Fischer, for one, addresses how communities grow a shared context [9].

Another issue regards the choice of language. A representation's ability to convey thought is determined by the language used and that language's compatibility with the languages of the thoughts themselves. For example, if a designer wants to convey a vision of an elegant water pitcher, a sketch's visual language would be appropriate, whereas a prose description would be unwieldy. The language must also provoke the recipient's thoughts in a useful way; a mathematical description of the pitcher's curves might be accurate, but could be difficult to understand (unless the recipient had a strong grasp of the equations used).

2.4 Discussion

By necessity, the description of our framework has been terse. Books have been written on topics to which we can only devote a few lines and entire research areas have emerged to study topics such as knowledge representation, effective communication, or creative thought. We have attempted to cite key contributions

to the general design literature to provide a starting point for those interested in more in-depth pursuits.

The primary message to take away from this section is that our framework provides a mechanism for bringing together existing design theory in a perspective that spans design disciplines. Each discipline must address the same product, process, and communication factors, in a manner suitable for its particular domain.

We must also remember that while the design process is essential, we should not lose sight of the design product defined in Section 2.1. The materials available to a discipline determine the space of its feasible outcomes, while a community's knowledge can build up a concept of the kinds of designs that tend to be desirable. Both these have a strong influence on how the design process can proceed. Software is no exception.

3. Tensions

Before we apply our framework to software design, it is useful to examine the framework from the perspective of tensions: elements in the framework that must serve multiple roles. As an example, a representation might be made by a designer to quickly make note of something they do not want to forget. In such a case, simplicity of creation is of the utmost importance to avoid interrupting the thought process. But if a representation is meant to illustrate a complex mechanical design, then faithfulness to reality becomes a dominant concern. A representation for communication might instead demand precision and understandability.

The implication is that different representations (and thus different languages) are needed to support these roles. Similar tensions govern the other elements of our framework. We recognize the following as the primary roles they must serve:

- *A goal constrains desirable outcomes.*
- *A goal guides generation of ideas.*
- *Knowledge informs outcome feasibility.*
- *Knowledge informs outcome desirability.*
- *Knowledge guides generation/evaluation of ideas.*
- *Knowledge guides creation of representations.*
- *Ideas constrain still possible outcomes.*
- *Ideas shift goals.*
- *Ideas build knowledge.*
- *Representations record personalized information.*
- *Representations communicate information.*

There is one additional role to discuss. Consider a graphic designer who sketches a logo, views the sketch, and then redraws it to hone in on the desired effect. In this case, the representation is not simply embodying an idea, but providing insight into the outcomes that the idea leads to. This feedback is described by Schön as a reflective conversation with materials [5], where a representation can provide insight beyond that possessed by the initial creator. Ideally, the designer will engage in "reflection in action", considering the representation without interrupting the creative process. This requires that the sketch, model, or prototype provide feedback in a form that is compatible with the language of the designer's goals, guiding him or her intuitively.

Returning to the roles and inherent tensions, the underlying issue is one of language. A given language can be precise, robust, nuanced, or easy to express, but rarely does a single language possess all of these qualities. Thus, when a given type of information must be used for several purposes there is a difficult decision to be made. Do we use the same language for multiple purposes, when it does not necessarily meet the needs of each? Or do we use a multitude of languages, and accept the burden of translating among them? A rich portfolio of languages with carefully considered tradeoffs, as well as tools for understanding and translating them, stands to advance any design field.

4. Software as a Unique Design Discipline

While Sections 2 and 3 introduce design factors that are common, most disciplines also give rise to unique challenges because of intrinsic difficulties exhibited by their context. In this section, we use our framework to highlight three such difficulties for software.

4.1 Large Space of Feasibility

In most design fields, the real-world properties of materials that are available to realize designs (i.e., building materials, art supplies) constrain possible outcomes significantly. It is impossible to have two parts in the same place; a brick cannot just float in space; and ink can only lead to certain effects on paper.

But software is virtual, and does not play by physical rules (other than needing to be run on a computer). The only constraints on it come from the programming languages used, which tend to be Turing complete and thus provide few constraints of their own. In terms of our framework, software design is faced with an enormous space of feasibility that does little to constrain still-possible outcomes. The success of a design is therefore determined almost solely by the space of desirable outcomes, which leads us to the next point.

4.2 Goal Specificity

A designer is usually given broad authority with respect to a particular design project. Constraints are applied, but these tend to be more guiding in nature (e.g., "we need a new building with 200 two-bedroom apartments" or "we need a new movie set representing a 1960s street"). While a designer needs to be keenly aware of these objectives, and generally will use early mock-ups and models to verify whether their plans resonate with the customer, they have a great degree of freedom in organizing the design and its details.

Initially, it seems like this could be true for software as well. An initial conversation between a designer and customer lays out a broad vision for what the software is to do. But, because software must fit into an existing environment (Brooks' conformity [10]), this vision has to be augmented with numerous details regarding how the software is to fit the existing manual and automated processes, as well as overarching business practices.

Put in terms of our framework, project desirability is governed by a set of very precise criteria that establish a tiny, idiosyncratic set of desirable outcomes. Combined with a large space of feasible outcomes, the designer must find the proverbial needle in an enormous haystack.

4.3 Language Gap

In most design fields, design decisions can be readily correlated to their outcomes. An architect deciding to add an archway draws the archway on paper or models it in the mock-up. Remove a note from a score, and it is visibly gone. There is a one-to-one correspondence between these decisions and the effects they will have.

Software design, however, experiences a significant language barrier. The designer's languages, involving software architectures, class diagrams, and the like, do not readily translate into outcome-level changes. When a designer adds a component, it is not immediately clear how this addition alters the space of still-possible outcomes. And when one considers software design languages more closely tied to outcomes, such as user interfaces drawings and behavior specifications, the opposite effect occurs: while decisions now clearly cut the space of still-possible outcomes, it is not clear how these relate to a class diagram or implementation plan.

This difficulty is related to Brooks' concept of invisibility [10], but our framework grounds this in a broader design perspective. The language of desirability (often English, use cases, or scenarios) is drastically different from the language of feasibility (programming languages) and translating between them is arduous. This, then, results in little support for Schön's reflective conversation with materials and also hinders Norman's concept of using design languages to communicate with the user [11]. Our goals are unable to serve their role of guiding our ideas and representations.

4.4 Summary

However brief, this discussion reveals software design to be a very difficult design discipline. Of course, this has been observed before. However, the strength of our discussion is that it grounds these observations in general design theory. In this paper's remaining sections, we will similarly ground specific software research contributions, allowing us to explain how they relate to software's unique challenges.

5. High-Level Approaches

When our community talks about design, it is often discussed in terms of its role in the software life cycle. While our framework does not address lifecycles directly, since it focuses on the individual designer, it has a lot to say about the *consequences* of such choices.

Consider the ubiquitous waterfall model. It directly prescribes a "design phase", and regards this phase as containing most, if not all, of the process' design thinking. The goal is articulated in a detailed requirements document, and the result of the design process is to be a plan for implementation. Purists, indeed, think of the requirements as an abstract specification of "what is needed", pushing all consideration of "how can it be realized" into the design phase.

In many ways, the waterfall model focuses on spanning the software language gap. It starts with a precise representation of a goal, which is translated over multiple languages before arriving at a final outcome. The representations used are often passed between developers, and because of the goal specificity of software, the languages used must be extremely precise. The emphasis throughout the process, and thus throughout the design process as well, is therefore on the faithful recreation of the ideas of a previous representation.

One effect of this approach can be found in the design process's goals. In Section 3, we explained that a goal both "constrains desirable outcomes" and "guides generation of ideas". While a requirements document constrains, it is intentionally bereft of inspirations for how design should proceed, providing little support for creative design. Representations are similarly focused, emphasizing their "document goals, knowledge and ideas" role, rather than supporting intuitive reflection in action. The waterfall model is not all bad; it is able

to translate goals into outcomes under software's uniquely difficult circumstances. But its approaches are often stifling to the creativity of design.

Agile methodologies' perspective on design can best be summed up by the phrase "the design is in the code" [12]. Such approaches advocate the rapid creation and evolution of code, treating it not as just the material in which the final program is implemented, but also as the representation in which programmers (each of whom is also a designer) express their design ideas. Unlike the waterfall model, agile approaches primarily use a single language, rather than translating between several.

Placing agile methodologies in our framework, we observe that the *message* of the approach ("freely play with code structures", "goals are broad understandings of user needs", "incremental releases for frequent designer-user contact") in theory is in line with our framework's concepts of idea generation and reflective conversations with materials. But we see that, in practice, these benefits are stifled by the use of code as a design representation. Programming languages are intended for implementation, and are not well-suited for communicating ideas between designers, nor recording ideas for one's personal use. Furthermore, without an intermediate means for organizing one's thoughts, the language gap between a designer's goals and representations of ideas for meeting them can be daunting.

We can analyze other life cycle approaches in similar ways. The Structured Analysis and Design method [13] provides an interesting example in its choice of language: Data Flow Diagrams smoothly span requirements, design, and database-based implementations. Open Source strongly emphasizes communication, with e-mail, bug, and repository archives providing representations of ideas and goals via a multitude of informal languages. The Problem Frames approach [14], meanwhile, focuses on connecting context-driven goals to high-level designs through shared knowledge.

Space prohibits us from discussing these approaches in more detail. But the message of this section remains: choosing a given approach has drastic consequences for many aspects of the design process. Even though different life cycle models seem to present completely disparate definitions of design, their perspectives can actually be precisely expressed and compared in terms of the tradeoffs that our framework describes. By leveraging the roles identified in Section 3, we can discuss them from a common basis and engage in objective comparisons of relative strengths and weaknesses.

We can also observe that no life cycle approach has been designed explicitly with a total design perspective in mind. It is compelling to envision such an approach. Some have put forward a notion of programming as design [12, 15]; architecture has been proposed as supplanting requirements [16]; and some have put forwards the beginnings of such a vision [17]. There are many signs that the role of design in software development may be broader than the community had first supposed.

6. Analysis and Agenda

The ideas, goals, and representations of a design project are specific to a particular customer's needs. But languages, tools, and knowledge span projects, and can be developed by a design community to shift the kinds of activities that are possible. In this section, we examine the progress of software design in terms of these categories and suggest several promising research directions. Some of these suggestions are unique, while others have been proposed before (for example, [18, 19]) but we provide new insight by presenting each in terms of our framework. We note we provide only a small subset of possible suggestions, ones we deem most important. Others can be derived from our framework in similar fashion.

6.1 Software Design: Knowledge

A designer's knowledge will fundamentally shape the decisions that they make. By building and sharing design information via research and trade magazines, conferences, web sites, and other educational venues, a community can improve the knowledge of its designers, and therefore the practice of its discipline.

But because of software's large space of feasibility, there are few rules, facts, or guidelines that can be universally applied. Knowledge is usually described in the context of a sub-domain defined in terms of the *implementation* technique used. For example, heuristics such as coupling and cohesion, database normal forms, and information hiding each serve as knowledge about how to design at the implementation level, given a particular approach to programming a solution.

There are also examples of early steps towards dividing our knowledge according to the way that the software will be *used*, as opposed to implemented. Product line architectures are promising in this regard. They describe a family of programs, all of which have a strong overlap in their desired qualities, and promote design decisions to be made at the goal level, in terms of features and options. Meanwhile, domain architectures have been developed for avionics systems [20]

and HCI researchers have begun to develop their own pattern language [21].

But in some ways, these approaches do not go far enough. The needs of different software systems are often so varied that there is little to relate them in terms of shared desirability, or shared possible outcomes. For example, there is little outcome-level knowledge that can be applied to the creation of both a document processing program and an immersive game. But *within* each of these categories there are consistently desirable qualities for its designers to strive for, and each should have its own set of principles, patterns, and frameworks to describe how to achieve them.

Research Direction 1: We should focus on desirability-based sub-domains, thus supplementing our existing, discipline-spanning knowledge and tools.

Another obstacle to broadly applicable knowledge is software's goal specificity. Because subtle details can determine whether a given outcome is desirable, it can be difficult to find an existing solution that meets a project's needs. Many aspects of software knowledge address this by considering the problem at a high level of abstraction, and allowing the designer to fill in the necessary details to make the knowledge "fit".

For example, architectural styles do not provide the designer with a complete solution, but rather a framework in which a project's specific needs can be met on an architectural level. This helps provide some structure on the otherwise vast space of feasible software designs. Design patterns work similarly, though at a lower level of abstraction, allowing class-level design to adjust for the project's needs. In addition, each pattern has a description of the situations in which it should be used, and a set of patterns is usually intended for a given programming language paradigm. In this way, patterns work within feasibility spaces.

These contributions perform admirably in overcoming software's difficulties, but each is very formal, providing advice in terms of rules and structures. Principles and patterns can be useful, but require very conscious application to a situation. An important element of knowledge, as mentioned in Section 2, is *aesthetics*, which can instead provide intuitive guidance during idea generation, reflective conversations with materials, and communication about design decisions.

In most fields, continued exposure to design solutions (such as famous buildings or well-designed products) allows designers to gain a sense of the space of desirable outcomes, and thereby a sense of good design decision-making. But because of the language gap in software, it is difficult to trace the security, speed, or stability of a high-quality piece of software back to the design decisions one would make in order to emulate it.

Thus, a software sense of aesthetics has to operate at a different level, and we will need to develop a feeling for intuitively evaluating artifacts at the level of our decision-making. This means evaluating and debating class-level diagrams, architectural drawings, and other documents, rather than simply the final programs that have been created. This is not easy, especially because aesthetics are intuitive, not coded, but a strong sense of aesthetics has proven key to shared knowledge and improved design quality in other fields.

Research Direction 2: Our community must begin to develop a sense of software aesthetics.

6.2 Software Design: Languages

Over the course of a given project, a variety of languages will be needed to allow a designer's representations to serve the variety of roles demanded of them. Unfortunately, software engineering's languages fall short in some regards.

Software's main strength in this regard is in communicating concepts unambiguously. UML and formal architectural description languages (ADLs) have rigid syntax, reducing their ambiguity. In agile approaches, even programming languages take on the role of design languages, being used to communicate certain implementation-level ideas unambiguously. Given that we have little shared knowledge that can span all of software design, such precision can be helpful.

But precision is not always desirable, and in fact can be arduous to specify. Two of a representation's roles are recording information for one's own use and communicating with others. In the former case, scant precision is needed, and in the latter, shared knowledge can be used to fill in missing details. But the majority of software design research has focused on symbolic languages, which require great precision.

Analog languages [22] (such as sketches or scale models) can be used at varying levels of detail and are vitally important to design. The use of pseudocode and whiteboard sketches is prevalent, but we have scarcely studied the ways in which such approaches are used, nor have we attempted to investigate and develop additional such languages. And while there is little need for full standardization, helpful conventions and guidelines should be researched. Such work will provide us with flexibility in our exploration and communication.

Research Direction 3: We must investigate ways to provide and effectively support decidedly less formal, analog design languages.

Design languages fall into a spectrum, addressing goals, ideas, or a little of both. They might describe a desired outcome; they might describe a feasible way to achieve it. It is important, however, that they continue to work towards addressing the language gap, in one way or another.

An intriguing notion is to avoid putting the responsibility of reducing the language gap on design languages, but to attempt to address it at the level of our materials: our programming languages. If these languages were constructed so they were better suited to design processes (i.e., defined a clearer space of feasibility that was more easily carved with design decisions), design would be simplified. Fourth generation languages allow for some of this effect in the database domain; we should explore other languages as well.

Research Direction 4: We need to become involved in the definition of new programming languages so they better suit our needs as a design discipline.

6.3 Existing Points of View: Tools

Tools serve two main roles in the design process: supporting the creation of representations and supporting their interpretation. Software design finds its creation activities reasonably well supported; a variety of design tools, such as Rational Rose or TogetherJ, allow models to be created graphically. Some tools may allow designs to be derived from source code via reverse engineering or support aspect-oriented design.

But the true strength of software's tools lies in their ability to assist in interpretation of design representations. Automated analysis tools can evaluate a design and provide insight into their goal-level qualities. Critics, such as those employed in ArgoUML [23], take this idea a step further and analyze design representations in real time. They then provide advice to the designer about improving the design, as expressed in terms of tool-level actions. From the perspective of our model, critics and intelligent agents [24] can almost be considered as co-designers, employing the knowledge and goals that they have been configured with, and interpreting representations accordingly.

In these tools we can see the hints of software approaching a true reflective conversation with materials, but so far, research advances have fallen short. In particular, a candidate tool must provide feedback about the qualities of the design that is: (1) presented in real-time, and (2) presented in a form that can be interpreted without interrupting the process of creation. Modern critics come close to achieving this, but fail on the second criteria. When a designer must stop their creative thought to read and interpret advice about their design, they are unable to reflect-in-action.

We propose researching tools that create "active" representations that change their appearance to reflect the qualities they posses. For example a visual design tool might adjust the colors of diagrammatic elements based on their role in the design. Alternately, the appearance of the lines connecting elements could change to illustrate their emerging relationships. Feedback of this kind could be selectively ignored, but would allow the designer to focus on areas whose appearance reflected qualities they deemed undesirable.

Such a tool might even instead utilize a physical metaphor, causing elements to shrink, stretch, or attract each other according to design's high-level properties. Concepts such as coupling, cohesion, or the number of functions in a class might be represented by physical concepts such as tension, hardness, or weight. Based on these attributes, the design would be modified by a "physic engine", providing the designer with real-time feedback through analogous, familiar phenomena.

Countless possible variations of active representations of this kind exist, and only experimentation will help us to understand which are helpful and which are not. But if the feedback can be sufficiently intuitive and unobtrusive, we see great potential for this approach.

Research Direction 5: We should develop tools that truly support a reflective conversation with materials through real-time, integrated feedback.

Software-based tools can also support a representation's communication role, acting simultaneously as a speaker and interpreter of design languages. CSCW researchers have made numerous interesting advances in communication tools, creating some that can support collocated as well as remote collaboration [25], or providing technological support for gesture interpretation [26]. But while this community has recognized the value of face-to-face, simultaneous communication [27], the software community has seen little tool support for it. Given the endorsement such communication draws in the general design literature [8], we should encourage more co-located design.

For example, simply working with others at a whiteboard is a vital design activity [28] and there are increasingly opportunities for tool support via electronic whiteboards. For example, a program that allows designers to intuitively organize multi-modal sketches could serve to support collective brainstorming. Or, by allowing multiple designers to independently modify a design on different parts of a board, and then compare and reconcile the differences among

them, conversation about ideas and assumptions might be spurred.

Research Direction 6: We should develop tools that support face-to-face design collaboration, perhaps utilizing new and emerging hardware options.

7. Discussion and Conclusions

It is difficult to evaluate a research contribution such as ours; the benefits provided by our work cannot be measured in terms of productivity increases, feature lists, or user survey results. Instead, we must ask "Does our work provide a useful perspective on software design and ways to improve it?" We believe our framework does exactly so by presenting beginning answers to the fundamental questions posed in our introduction.

We asked about the critical factors at play in software design and find they can be described as the elements of our model: knowledge, goals, ideas, and representations, plus activities, languages and, tools. By considering these factors in terms of software's specific needs, as we did in Sections 4, 5, and 6, we gain valuable insight into software's nature as a design field.

We also explained some of software's strengths and weaknesses by discussing existing software design contributions and using our framework to describe their roles. We have shown that seemingly disparate advances serve to address common concerns, such as gathering feedback about the outcomes of our designs or guiding us towards more effective solutions.

Finally, we suggested directions for further research. While we cannot yet be certain of their value, given the theoretical basis and the fact that advancing in software design research to date correlate with the lessons of our framework, we have great confidence in their potential.

In many ways, software design has made important progress. We are addressing software's unique difficulties, and our languages, tools, and representations serve certain roles effectively. But, for our discipline to mature, we must begin to focus our efforts on those areas in which software designers are ill-supported. This will require principled research and debate into all factors surrounding the software design task.

It is very important to note, though, that we do not consider the answers provided in this paper to be definitive, nor final. We have presented a new framework for discussing design, one that itself must be considered, debated, and refined. Further, lone researchers cannot determine the state of software design, nor its directions forward. Rather, we hope we have provided the community with a seed for extensive debate. It is only through presentation, evaluation, and refinement of explanations of software design that we can advance our understanding of its nature, and of its practice.

8. Acknowledgments

Effort partially funded by the National Science Foundation under grant numbers CCR-0093489, DUE-0536203, and IIS-0205724. We thank the anonymous reviewers of a previous version of this paper.

9. References

[1] A. Newell and H. A. Simon, "GPS: A Program that Simulates Human Thought," in Computers and Thought, McGraw-Hill, NewYork, NY1963.

[2] T. Smithers and W. Troxell, "Design is Intelligent Behaviour, but What's the Formalism?," AI EDAM, vol. 4, pp. 89--98, 1990.

[3] H. A. Simon, *The Sciences of the Artificial*, 3rd Ed., MIT Press, 1996.

[4] J. C. Jones, *Design Methods.* John Wiley and Sons, New York, Inc., 1970.

[5] D. A. Schon, *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, 1983.

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

[7] M. Stacey and C. Eckert, "Against Ambiguity," Computer Supported Cooperative Work, vol. 12, 2003.

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

[9] G. Fischer and J. Ostwald, "Knowledge Communication In Design Communities," in Barriers and Biases in Computer-Mediated Knowledge Communication. Springer, New York, NY.

[10] F. Brooks, *The Mythical Man-Month.* Addison-Wesley, Reading, MA, 1995.

[11] D. A. Norman, *The Design of Everyday Things.* Basic Books, NewYork, NY, 2002.

[12] K. Beck, *Extreme Programming Explained.* Addison-Wesley, Reading, MA, 1999.

[13] T. Demarco and P. J. Plauger, *Structured Analysis and System Specification*, Prentice Hall, 1979.

[14] M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley Professional, Reading, MA, 2000.

[15] A. F. Blackwell, C. Britton, and e. al., "Cognitive Dimensions of Notations," Cognitive Dimensions, 2000.

[16] R. N. Taylor, "Requirements Engineering is SO Twentieth Century," Keynote at STRAW, 2001.

[17] T. Winograd, *Bringing Design to Software.* Addison-Wesley Professional, Reading MA, 1996.

[18] T. Winograd, "From Programming Environments to Environments for Designing," Communications of the ACM, vol. 38, pp. 65 - 74, 1995.

[19] G. Fischer, A. Girgensohn, K. Nakakoji, and D. Redmiles, "Supporting Software Designers with Integrated Domain-Oriented Design Environments," IEEE Transactions on Software Engineering, vol. 18, 1992.
[20] D. C. Batory, Lou; Goodwin, Mark; Shafer, Steve, "Creating Reference Architectures: An Example from Avionics," Software Engineering Notes, 1995.
[21] J. O. Borchers, "A Pattern Approach to Interac-tion Design," AI & Society, vol. 15, pp. 359-376, 2005.
[22] Ö. Akin, "Variants in Design Cognition," in Knowing and Learning to Design, 2002.
[23] J. Robbins, Hilbert, D. and Redmiles, D., "Argo: A Design Environment for Evolving Software Architectures," Nineteenth International Conference on Software Engineering, Boston, MA, 1997.
[24] T. Mandel, " Social User Interfaces and Intelligent Agents," in The Elements of User Interface Design, Wiley, 1997.
[25] M. Roseman and S. Greenberg, "TeamRooms: Network Places for Collaboration," Proceedings of the 1996 ACM Conference on Computer supported Cooperative Work, Boston, MA, 1996.
[26] M. M. Bekker, J. Olson, and G. Olson, "Analysis of Gestures in Face-to-Face Design Teams Provides Guidance for How to Use Groupware in Design," Proceedings of DIST, New York, 1995.
[27] ACM, "Workshop on Shared Environments to Support Face-to-Face Collaboration," CSCW, 2000.
[28] S. W. Ambler and R. Jeffries, *Agile Modeling*, Wiley, 2002.