Software Architecture Design From the Perspective of Human Cognitive Needs

Jason E. Robbins
Information & Computer Science
University of California, Irvine
Irvine, CA
(714)842-7308
jrobbins@ics.uci.edu

David F. Redmiles
Information & Computer Science
University of California, Irvine
Irvine, CA
(714)842-3823
redmiles@ics.uci.edu

ABSTRACT
Software architectures are useful, in part, because they use the appropriate level of abstraction to support the design of complex systems. Software architecture research has quickly evolved to the degree that design environments have been implemented to support software architects in creating new designs. We report on a software architecture design environment named Argo that differs from other approaches by paying attention to the human, cognitive needs of software architects, as much as to the representation and manipulation of the architecture itself. We emphasize the primary considerations by contrasting the human cognitive design process to the systems-oriented software design process. Human-centered features in Argo focus on the application of critics for providing design feedback, design processes for supporting critics, and multiple architectural perspectives for aiding human designers.

Keywords
Design environments, critics, software architectures, architectural styles, human-computer interaction, human cognition

INTRODUCTION

Much attention in software engineering research today is focused on the use of software architectures in design [1, 2, 15, 24]. The major motivation is that software architectures provide the appropriate level of abstraction to support the design and understanding of complex systems. Furthermore, paralleling the research in software reuse, the research in software architectures has the potential of yielding high quality products with minimal effort. The research has quickly reached the point where design environments have been implemented to support software architects in creating new designs by combining components within architectural styles. For example, Garlan and colleagues have explored the design environment component requirements for supporting the definition, storage, retrieval, and application of architectural styles [9].

We too have been motivated by the potential benefit of using higher level abstractions in architecture design and, more generally, by the appealing arguments of augmenting people's ability to solve design problems [3]. We have built a software architecture design environment, called Argo[1], to support the design of human-computer interface software. The design environment components explored by Garlan and others provide an essential basis. However, in building Argo, we have also adhered to the issues raised in studies of the human cognitive needs for performing design.

We view software architectures as the unifying structures of software systems. They are the "big picture" of a design, and the basis for the designers, implementors, users, and maintenance programmers to understand a system. This characterization of software architecture emphasizes the human cognitive needs of stakeholders in software design. In searching for an architectural approach to software development, we must keep human cognitive needs central to our choice of paradigms, languages, and tools.

Our approach to building a better software architecture design environment is to identify and address human cognitive needs that arise in architectural design. Specifically, we examine three major cognitive theories describing people's behavior in design situations. These theories explain the problems human designers will face when working within the context of software architecture design. In particular, the theory of *reflection-in-action* suggests human problems that may be ameliorated by the design feedback provided by critics. The theory of *opportunistic design* has implications for how process representations should be used in a design environment. Finally, a theory of *comprehension and problem solving* suggests the need to present information from multiple perspectives.

Each of these topics is examined in detail. We begin by contrasting human cognitive design processes with software design processes and hypothesize how the two may be

This research is supported in part by the Air Force Material Command and the Advanced Research Projects Agency under Contract Number F30602-94-C-0218. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred. Additional support is provided by Rockwell International.

1. Argo was the name of the ship that Jason sailed in Greek Mythology. We hope our Argo will aid software architects in navigating design spaces.

Table 1: Cognitive Theory / Design Environment Components

This Table outlines the relationship between the cognitive design theories reviewed in the paper, their implications for requirements on design environment components, and their support as implemented in the Argo design environment.

	Cognitive Design Theories	Design Environment Components	Features in Argo
Feedback	Reflection-in-Action	Construction, Critics: Expert Design Feedback, Types of Feedback, Management of Feedback	Graph Editor, Representation of Critics
Process	Opportunistic Design	Cognitive Design Process: Flexibility, Visibility, Reminding, Delaying Detail, Timeliness	Tailorable Design Process Model, Design Process Perspective, To Do List
Perspectives	Comprehension and Problem Solving	Multiple Design Perspectives	Design Process Perspective, Conceptual Architecture, Execution Perspective, Modular Perspective, …

reconciled. We discuss details of the proposed features in terms of components for a design environment. We illustrate the implementation of these components in the Argo design environment for software architectures. The paper concludes with a brief discussion of related work and summary.

HUMAN, COGNITIVE NEEDS IN DESIGN

Design environments must serve the cognitive needs of the designer. But what exactly are the cognitive needs of a person faced with a new design[2] task? In what ways does the designer proceed to work through a design problem? In this section we attempt to provide a comprehensive answer to these questions by examining three major, cognitive theories or schools of thought about human design and problem solving. Each theory or school of thought sets a different priority on a set of cognitive issues.

These are not prescriptive theories (what design should be) but rather, they are descriptive theories (what designers have been observed doing). One goal of our work, and others striving for good design environments, is to reconcile how design *is* with how design *needs to be*. In this vein, we follow-up each presentation of a cognitive theory with a discussion of its relation to software engineering processes and indicate which components of the Argo design environment are included to address these issues. Table 1 outlines the main points of this discussion; each cell in the table is discussed in one subsection of this paper.

Reflection-in-Action

Donald Schön coined the term "reflection-in-action" to describe the behavior he observed of building architects working on new designs [19]. The basic principle was much like that of prototyping in software engineering. Designers *name* or identify the elements of a problem. They *frame* or organize these elements into a proposed solution. They *act* on this proposal, creating a sketch, model, or constructed artifact. In the act of bringing the proposal to this new level of realization, they experience a *breakdown* or fault in the design. Breakdowns force designers to reflect on their design, and in particular, force them to iteratively re-frame their proposed solution. Schön later referred to the process as a "conversation with the materials of design [20]." This theory of design is further substantiated in Polanyi's work on *tacit knowledge* [16]. The theory demonstrated by Polanyi is that peoples' ability to recall or apply knowledge is possible only when they are in the situation of acting. Guindon, Krasner, and Curtis noted the same effect as part of one study of software developers [11]. Calling it "serendipitous design," they noted that as the developers worked with the design, their mental model of the problem situation and hence their design improved. Fischer, et. al. [6] characterizes this process as one of *co-evolution*: designers simultaneously refine an artifact being constructed and their understanding of that artifact.

However, there are inherent dangers in this "natural" design process. It is common knowledge among software engineers that processes like prototyping allow artifacts to rapidly grow out of control. Inconsistencies evolve and other requirements are easily overlooked. Furthermore, Schön

2. Here, *design* does not refer to the traditional modular design phase of system development, but rather to the activity of making complex things in general. We view architecture as a special case of the broader issue of design, and use *design* and *designer* interchangeably with *architecture* and *architect*.

noted that the process of reflection-in-action is most successful when designers can draw upon knowledge about the design components accumulated over many years. The process is least effective when the designers are faced with using new or unfamiliar materials. Software architects are frequently faced with applying new software components and architectural styles.

To reconcile the naturally observed design process of reflection-in-action with the practical needs of a software engineering process, we adapt the notion of critics first proposed by Fischer and colleagues [7]. A *critic* in this context is an agent that monitors a human designer's progress in refining a design. Each critic reacts to violations of a specific design rule. The critic's rule can encode a hard constraint that may not be violated in the final design or a soft constraint that represents a suggestion or rule of thumb. Their general role is to allow the observed design process of reflection-in-action but to maintain the positive properties of a prescribed software process. In particular, they augment a human designer's ability to detect potential breakdowns, especially in the situations where designers are working with unfamiliar components. Critics augment designer's abilities by delivering knowledge accumulated over past designs or by more experienced designers.

Opportunistic Design

Opportunistic design is a theory that developed out of empirical observations of mechanical and software engineers [11, 25]. The theory focuses on the observation that designers frequently start with a hierarchical, goal-oriented plan for solving a design problem, but deviate from that plan, performing goals and sub-goals out of order. The theory explains the deviations in terms of *cognitive cost*. Roughly, when one sub-goal is more familiar or "doable" in the designer's mind than the current sub-goal, the designer selects the more doable, even if it means deviating from the order in the original plan [25]. The theory of opportunistic design builds on the cognitive research by Soloway and others on programming plans [22, 23]. In this context, programming plans represent the knowledge programmers use to develop problem solutions. More generally, they provide the basis for a notion of design schemas, which are used to identify what designers know or don't know about a design situation. Rist, for example, used design schemas in creating a simulation of designers' opportunistic behavior, examining in detail the cognitive expense or complexity designers face when creating new plans for unfamiliar situations [18].

In their empirical study, Guindon, Krasner, and Curtis observed several specific situations in which deviations from a planned or prescribed order occurred and created problems for a software design process. With respect to situations that cause deviations, they observed lack of application and solution domain knowledge; difficulty in allocating time and effort to activities; difficulty in selecting among alternatives; difficulty in considering constraints; inability to perform mental simulation of designs; difficulty in returning to subproblems after a previous deviation; and difficulty in merging partial solutions to subproblems. They categorized these breakdowns more generally as corresponding to designers' lack of knowledge about application domain or solution domain, lack of good process understanding, and breakdowns from the combination of the two.

Once again, our aim is to reconcile the natural or observed design process with the requirements of a software engineering process. In particular, many of the problems identified by Guindon, Krasner, and Curtis can be alleviated by a representation of the design process. Argo supports a process representation; however, in our approach, the process is not used to force designers to follow a rigid plan, but rather as a resource to support a more opportunistic design strategy. For instance, the process representation is used to support a checklist feature which helps designers return to deferred problems and goals. An overview perspective of the process model helps designers understand their progress and set priorities. Furthermore, designers may edit the process model and checklist, tailoring it to their current situation. We revisit this topic in "Support for the Cognitive Design Process."

Comprehension and Problem Solving

Based on studies of people solving word algebra problems, Kintsch and Greeno developed a theory of problem solving behavior [12] and later Fischer and Kintsch extended the theory to explain the behavior of software engineers [5]. The theory states that designers begin with a vague notion of a problem situation which they must successively refine into a precise statement of a design. The designer's mental model of the problem is called the *situation model* and consists of background knowledge and problem-solving strategies the designer knows and which are related to the current problem. The designer's mental model of the solution is called the *system model;* a correct system model allows the designer to write out a design, for example, in a specification or programming language. The theory emphasizes the comprehension aspect to design. For instance, experts have enough background knowledge and problem-solving strategies to identify, and then map, the correct elements of a situation model into a system model. Novices may have to form and reform their intermediate models many times before discovering the correct, situation and system models.

By emphasizing the role of comprehension in design, this theory raises the issue of how designers' mapping from situation to system model may be improved. Pennington examined the theory more closely with respect to the mental models designers need in order to develop correct situation and system models [14]. Her work indicated the usefulness of multiple presentations of design examples. For example, in the domain of programming, she evaluated presentations such as data flow, control flow, and functional decomposition. Redmiles generalized the notion of presentation to multiple perspectives in a system called Explainer [17]. Explainer provided its users with explanations of program code examples. In terms of the theory of opportunistic design, Explainer provided designers with design schemas

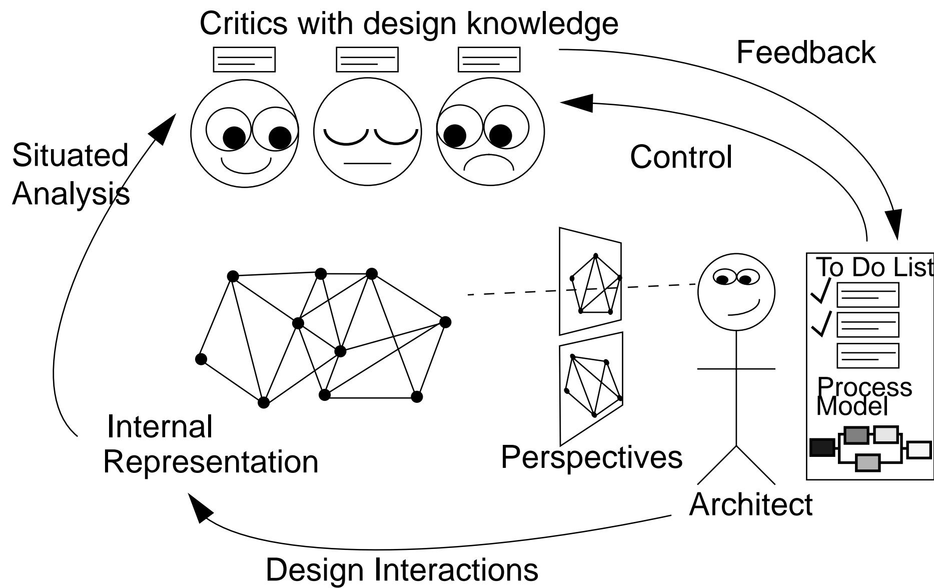

Figure 1: Selected Design Environment Components of Argo

they were lacking, thus reducing the cognitive cost of following a plan.

In Argo, multiple perspectives are again applied. These perspectives include conceptual component communication, component-to-module mapping, component protocol conformance, and module-process mapping. The goal is to aid comprehension of architectural designs based on the above considerations. The multiple perspectives also have an intuitive appeal. They support the separation of concerns in software design and help designers manage complexity.

COMPONENTS FOR SOFTWARE ARCHITECTURE DESIGN ENVIRONMENTS

A domain oriented design environment [6] is a tool which supports designers in the design process with domain knowledge. Figure 1 presents selected conceptual components of a software architecture design environment. In this section we discuss these design environment components and how they support the cognitive needs identified in the previous section.

In our approach, the architecture may be represented in an object model as connected graph. The designer views and manipulates that representation through various perspectives such as conceptual component communication and allocation of execution resources. Automated design critics in the environment monitor the design and deliver knowledge to the designer when relevance predicates are satisfied. Critics place their feedback in a "to do" list. The designer uses a process model as a resource in carrying out his design process, while the design environment uses that process model to ensure the timeliness of delivered knowledge. The process model also structures communication between the designer and the critics.

A software architectural style is a set of rules that constrain the architecture and provide guidance to the architect. Styles are based on a set of recurring patterns observed in a family of applications. Styles may also guide the design environment builder, in that critics and perspectives may be organized according to style.

Critics

As software architects edit and reflect on their designs, they develop them through many intermediate states, often many invalid states. Tools which do not allow rules to be violated (e.g., some syntax directed program editors) tend to be much more difficult to use than those that do (e.g., standard text editors). At the other extreme, tools which do not check the validity of designs as they are being entered give no guidance to the designer at the times when most decisions are being made. Design environments take a middle road: they critique the design as it is being entered. In the vast majority of cases critics simply advise the architect of potential errors or areas needing improvement in the design; only the most severe errors are prevented outright. Design feedback is managed so as to inform the architect with minimal distraction.

Architects can benefit from the knowledge of domain experts if that knowledge is delivered to them via critics. Rather than place all the burden of precision and restriction on the style designers, we assume that the software architect is capable of making final decisions regarding the application of the advice given. While some of the assumptions of software components, connectors, or styles are implicit, it is usually possible to make them explicit as rules, even if merely as rules of thumb. We anticipate that much of the practical, day-to-day knowledge about software architectures will take the form of guidelines or rules of thumb, and styles will accumulate so many of them that a non-disruptive feedback mechanism is needed. English grammar checking tools are in an analogous situation: some rules are too complex or fuzzy to implement precisely, and a typical critique produces enough feedback to overwhelm the user.

There are a variety of potential types of critics, and each type delivers a specific kind of knowledge. Correctness

critics detect syntactic and semantic flaws in the partial design. Completeness critics detect when a design task has been started but not yet finished. Consistency critics detect contradiction within the design. Presentation critics detect awkward use of the notation. Alternative critics remind the designer of alternatives to a given design decision. Optimization critics suggest better values for design parameters. These types serve to categorize the critics so that they may be understood and controlled as groups. Some critics may be of multiple types, and new types may need to be defined as appropriate for a given application domain.

Critics react to the design as it is being entered, before significant effort is invested to refine tentative design decisions. This tight integration of design feedback into the design process supports the cognitive need to reflect on the design described under "Reflection-in-Action." Because critics are active from the beginning of the design process, they can advise the architect as design alternatives are explored and decisions are made. Organizing design analysis into critics imposes the burden on the critic author that analysis must be possible on a partial design. However, this choice yields the advantage to architects that more information is available earlier. To reduce the burden on the critic author, design environments must manage critics and their feedback so that critics may be overly eager and pessimistic without distracting the architects attention.

Cognitive Design Process

Each of the three cognitive theories discussed above highlighted differences between cognitive design processes and software design processes. However, one of the most striking differences was raised in the discussion of opportunistic design. Designers deviate from plans, even their own plans. These deviations may be unavoidable or even desirable from a cognitive perspective, but they lead designers into a variety of difficulties as discussed in the Guindon, Krasner, and Curtis study.

We respond to observations of plan deviations by maintaining a representation of the design process that accommodates the observed cognitive processes. Design environments can address the cognitive needs of designers by focusing on certain design process characteristics: flexibility, visibility, reminding, delayed commitment to detail, and timeliness of feedback.

The foremost design process characteristic is flexibility. Designers must be allowed to deviate from a prescribed sequence and allowed to choose which goal or problem is most effective for them to work on. Designers must be able to add new goals or otherwise alter the design process as their understanding of the design situation increases. The plan serves primarily as a resource to the designer's cognitive process, and only secondarily as a constraint on it.

Design environments can support opportunistic design by providing visibility into the cognitive process and helping the architects orient themselves in the process. In particular, the design environment should be able to represent what has been done so far and what is yet to be done in defining the design. Visibility enables architects to deviate from their planned sequence. They may take a series of excursions into the design space and re-orient themselves afterwards to continue the design. Such excursions can lead to increased understanding of the design situation.

Design process representations should support reminding. Reminding helps architects revisit incomplete details or overlooked alternatives. Reminding is needed most when designs are complex, design alternatives are many, and design processes are loosely constrained.

Delaying commitment to design details is supported by the combination of flexibility, visibility, and reminding. If architects are forced to commit to each design decision that is begun before any analysis can be done, then the effort required to explore design alternatives will be much higher. Higher effort means that fewer alternatives will be considered, which reduces confidence in the design. Higher effort also shifts attention away from the design at hand and into planning required to use the tool efficiently. In our approach, the ability of critics to deliver partial critiques of partial designs in a managed and usable form allows designers to evaluate their designs without premature commitment to design details.

A final characteristic is timeliness of feedback. The task of a critic is to deliver information to aid in design decisions made as part of a design process. For the critics to produce timely information they must have an explicit model of the design process and the architect's progress in it. When the design process is modeled as tasks where each task addresses only a few design issues, then a representation of which tasks are in progress gives a strong indication of which design decisions are in progress and thus which critics are timely. Criticism will be distracting if it involves design decisions that the architect has not yet begun considering. The architect can also indicate when he considers a task finished, and the design environment can generate additional criticism at that time, perhaps marking the task as still in progress if there are high priority "to do" list items pending.

Multiple Design Perspectives

The architecture of a complex software system is itself a complex artifact. As that artifact is constructed and evolved, it must be understood by designers and implementors. Key to understandability of the design is the management of complexity. Dividing the complexity of the design into multiple perspectives allows each perspective to be simpler than the overall design. Moreover, separating concerns into separate perspectives allows information relevant to certain related issues to be presented together in an appropriate notation.

In any complex design the expectation of a single unifying structure is a naive one. Complex software system development is driven by a multitude of forces: human stakeholders in the process and product, functional and non-functional requirements, and low-level implementation constraints. In well architected systems there are unifying structures, such as code and data organization, intercomponent communication patterns, and naming

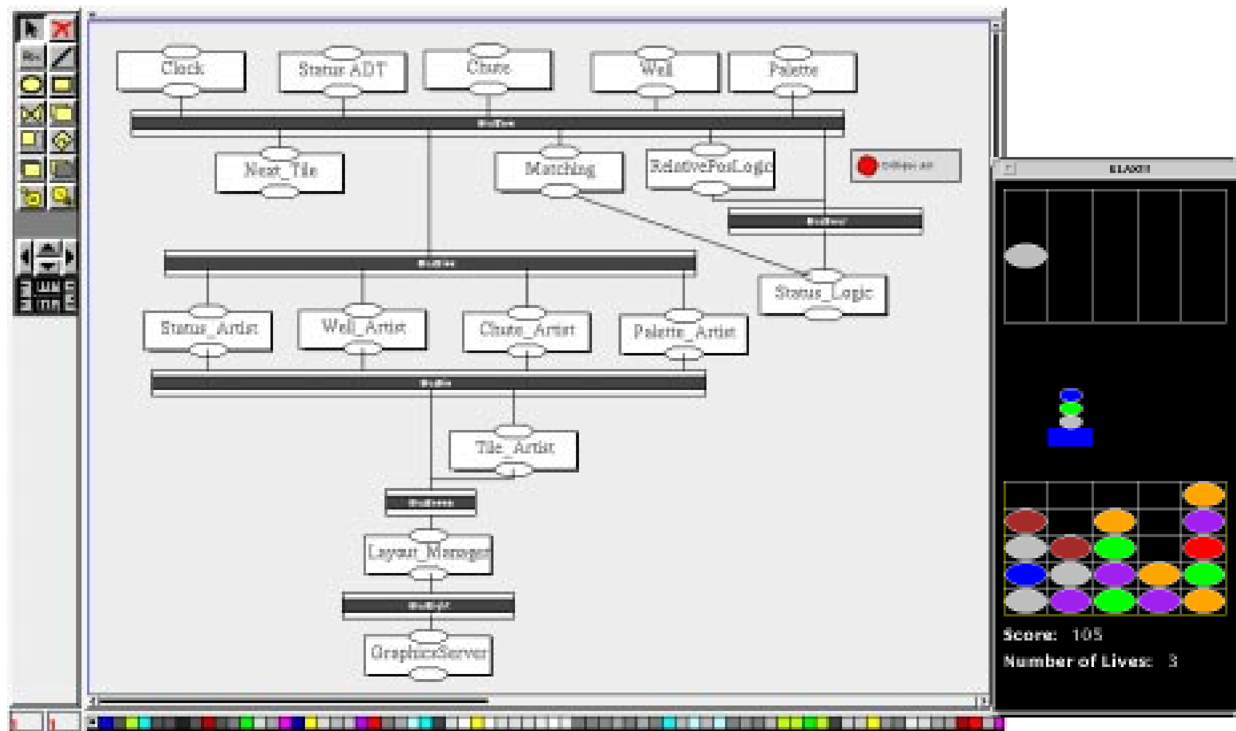

Figure 2: The KLAX Conceptual Architecture Perspective and Screenshot of Running KLAX System

conventions. However, complex software systems by definition will not have a *single* unifying structure. Furthermore, architects must work with other stakeholders, each of whom has their own task and background, and thus may need different perspectives.

The need for a wide variety of perspectives is evident in the wide range of high-level notations currently in use in software design. Each notation focuses on some aspects of the design and ignores others. Some typical perspectives include: the organization of code into files and directories, class hierarchies, data flow and communication pathways among components, division of the name space for identifiers via the use of naming conventions, and the run-time distribution of components over operating system processes and hosts. It is our contention that no fixed set of perspectives is appropriate for any possible design; instead perspective views should emphasize what is currently important in the project. When a new set of issues arrises in the design, it may be appropriate to use a new perspective on the design in addressing those issues. Although, for a specific domain we can identify some useful perspectives ahead of time.

SCENARIO OF DESIGN SUPPORT USING ARGO

The example used below is explained in [13]. In that paper, it is used to illustrate the C2 architectural style [24], developed specifically for use in the domain of user interface software. Here, the example is used to illustrate the interactive support Argo provides for software architecture design. Argo has been built with the C2 style in mind, although support for other styles is possible. The example system implements a simple video game, called KLAX[3], in which falling colored tiles must be arranged in matching rows or columns (Figure 2).

Software architects use Argo by placing graphical representations of architectural elements (e.g., components and connectors) in a drawing area and connecting them to make graph structures representing each of several perspectives on the system. For example, in Figure 2 conceptual components storing the state of the game (top row) have been connected via a message bus to conceptual components handling game scoring and matching logic (second row). After an element has been placed it may be annotated. For example, a component has attributes for its name and a list of services provided and requested through each of its two ports (having exactly two ports is a C2 style rule).

As architects make design decisions, automated critics in the design environment deliver knowledge relevant to those decisions. For example, Figure 3 shows the output of a critic that has detected a unfulfilled service request from the top port of the Status_Logic conceptual component[4], the error is caused by a spelling error in the name of the requested service. Architects rely on the design feedback when they

3.KLAX is trademarked 1991 by Atari Games.

4. In C2 unfulfilled requests are not necessarily errors. In Argo we provide a flag to mark requests which the component designer intends to *always* be fulfilled.

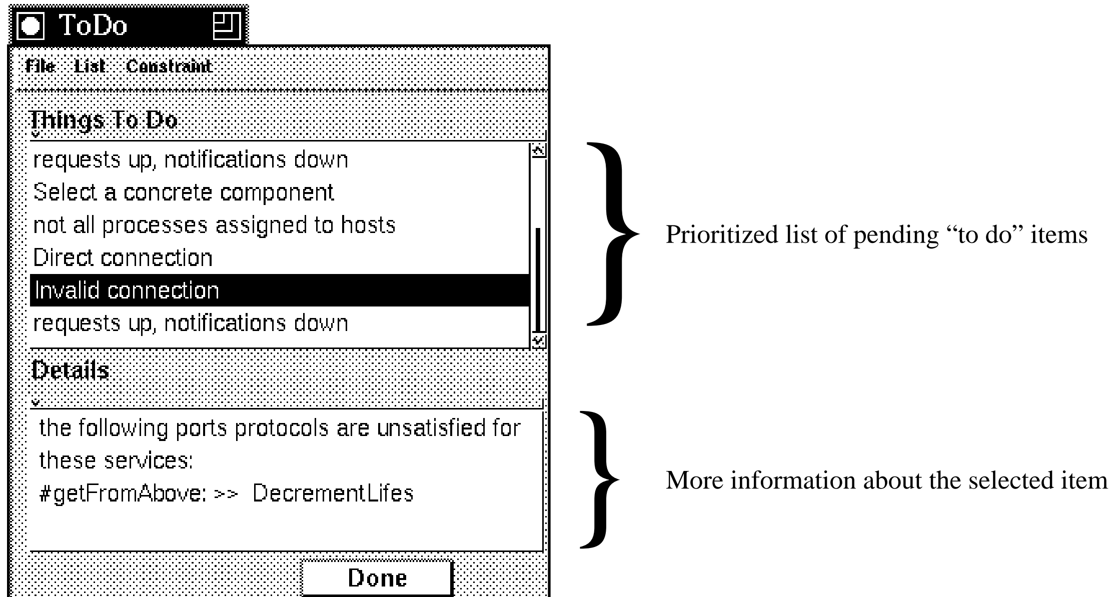

Figure 3: The Architect's To Do List with Feedback from Critics

Table 2: Selected Argo Architectural Critics

Name of Critic	Knowledge Type	Relevant Design Task	Explanation
Invalid Connection	correctness	checking	Mandatory message signatures not satisfied by adjacent components in the conceptual architecture
One Up One Down	correctness	checking	Violation of C2 configuration rules
Simpler Comp. Avail.	alternative	choosing	A "smaller" component will "fit" in place of what you have
Too Much Memory	consistency	profiling	Calculated memory requirements exceed stated goals
Need more reuse	consistency	choosing	Percentage of reusable components is below stated goals
OS Incompatibility	consistency	annotating	Components have conflicting environmental requirements

encounter some problem, do not know what to do next, want to consider alternatives, or are ready to solidify an architecture by resolving open issues. With new insight, the architects then continue to evolve the design. When a design task nears completion the critics remind the architect of unfinished details or unexplored alternatives.

A critic is represented as a piece of design knowledge and a predicate to determine when the critic should deliver that knowledge. The predicate is evaluated with additional checks to determine if the critic would be timely and relevant. When a critic is active and its predicate is satisfied, it delivers its design knowledge in the form of an item inserted into a "to do" list (Figure 3).

The "to do" list helps orient the architect in the design process by enumerating possible issues that may be addressed next. Further supporting the architect's cognitive processes is a simple process model that describes steps for using Argo to make a new design, and visually depicts the progress made at a given point in time (Figure 4). The design process perspective aids the architect in returning from excursions in the design space and determine the relevance of each critic. For example, the Invalid Connection critic is active because it has not been hushed, correctness critics have not been disabled as a group, and some process step that is marked with the decision category "checking" is currently in the state "in progress". Rather than try to automatically enact the process model, we rely on the architect to declare when each task is started and finished. The "to do" list and process model are intended to support architects, not control them.

Architects may create and edit designs in several coordinated perspectives (Figures 2, 5, and 6). Each architectural perspective is presented with a notation appropriate to that perspective. Architectural elements which are relevant to multiple perspectives are presented once in each of those perspectives in a form that is appropriate to each. Manipulating an element may modify

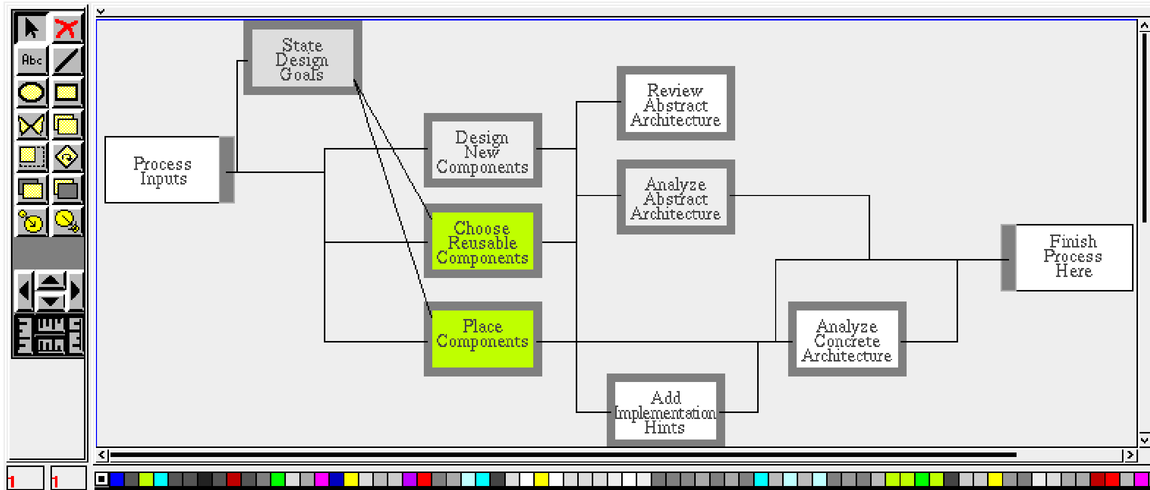

Figure 4: A Software Architecture Design Process Perspective

its state, which is reflected in all perspectives showing that element.

DESCRIPTION OF ARGO

This section describes the Argo software architecture design environment in detail. Features that support design feedback from critics, the human cognitive design process, and viewing the design from multiple perspectives are discussed in the following subsections. These features correspond to the cognitive needs and generic design environment components discussed in previous sections.

Argo stores architecture and process models internally as a connected graph with annotations on the nodes and arcs. The graphs are manipulated through a diagram editor, while the attributes are edited via dialog boxes. The various perspectives are simply projections of that internal connected graph. Since specific constraints on the graph are handled in the critics, Argo's infrastructure makes very few assumptions about the characteristics of that graph. It is this simple, precise, and flexible design representation that allows Argo to work with diverse architectural perspectives.

Support for Critics

Critics deliver knowledge to software architects about the implications of, or alternatives to, a design decision. Critics are agents which watch for specific conditions in the partial architecture as it is being constructed and notify the architects when those conditions are detected. They support a design process of action followed by reflection. The architect's ability to reflect on a design is enhanced by the knowledge encoded in the critics.

A variety of underlying components support the use of critics in Argo. First, Argo has a framework for implementing critics and a run-time facility for evaluating the predicates of active critics. Predicates are currently implemented as a combination of fragments of Smalltalk code and cross-reference tags. Second, Argo has a variety of user interface mechanisms for controlling which critics are active at a given time and for managing design feedback.

Table 3: Details of the Invalid Connection Critic

Attribute	Value
Name	Invalid Connection
Type	Correctness
Decision Category	Checking
Smalltalk Predicate	`[:comp		invalidServices	invalidServices := comp inputs , comp outputs select:[:s	s isSatisfied not]. invalidServices isEmpty not.]`
Hushed	False
Feedback	"The following port protocols are unsatisfied for these services:" <<a list of ports and services>>
Expert	jrobbins@ics.uci.edu

Table 2 gives a description of some example critics, while Table 3 presents one critic in detail.

Argo's framework for critics associates them with the architectural elements in the design; there is no central rule base of critics. When a new type of architectural element is defined, new critics are defined for it. Critics which cannot easily be associated with any one architectural element are represented as free standing critics associated with a particular perspective view. For simplicity, Figure 1 presented critics as external to the architecture representation looking down on it. A more literal presentation would show critics associated with each node of the architecture representation.

Argo currently has less than 20 critics, but future research prototypes or deployed design environments could have hundreds or thousands of critics. It is important not to overwhelm the architect with too much information. Feedback management techniques in Argo improve the capability of critics to trigger at the right time with the right information, and also help the architect make sense of

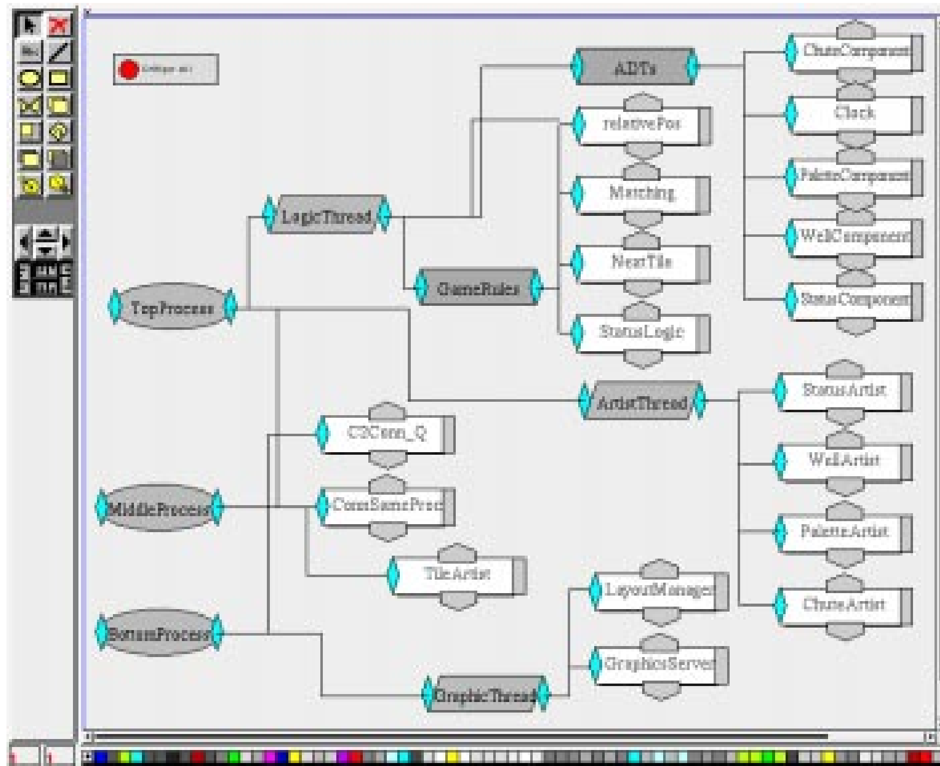

Figure 5: The KLAX Execution Perspective

criticism by organizing it. Critics may be active or inactive depending on the state of the architecture, the design process, and stated design goals. If the critics are providing inappropriate feedback or are otherwise felt to be too intrusive, the architect may *hush* them, rendering them temporarily disabled. Critics insert their criticism into the "to do" list, which is prioritized based on the severity of the criticism and the state of the process model. Architects may address issues in an order they choose and they may also reorder the list or insert items as reminders to themselves.

Support for the Cognitive Design Process

As discussed under "Opportunistic Design", the design process and the architect's progress in that process can be of great value in allowing the architect to choose which issues to address next, but that same process information can also be used by the design environment to ensure that design feedback is delivered in a timely fashion.

A model of the design process with progress information provides leverage in managing feedback. We use an IDEF0-like notation to model the process tasks involved in an typical design process (Figure 4). Each task in the design process works on input produced by upstream tasks and produces output for consumption by downstream tasks. No control model is mandated: tasks can be done in any order (provided needed inputs are available), tasks can be repeated, and any number of tasks can be in progress at a given moment. Each task is marked with a status: future, in progress, or done. Status information is shown graphically by the use of color in the process diagram. Each task is also marked with a decision category symbol: building, choosing, checking, annotating, or profiling. Decision category symbols and statuses are used to limit the activity of critics and thus avoid producing feedback that is not timely and relevant.

The design process model shown in Figure 4 is a fairly simple one, partly because the C2 architectural style does not impose any explicit process constraints, and partly because this example does not consider issues of organizational policy which might constrain or complicate the design process. In practice, the process would be more complex if these limiting assumptions were removed. For those reasons, the process of defining and evolving the process (usually called the meta-process) can itself be a complex, evolutionary task which could benefit from feedback from critics.

The process model in Argo is "first class," in that it may be visualized, manipulated, and critiqued just as the architecture itself may be. Argo's infrastructure supports manipulation of any artifact in the general class of connected graphs. Multiple perspectives may be defined to view the process as appropriate for various stakeholders. The architects may define, modify, and annotate the architecture via the same editor used to work on architectures. Critics may operate on the process model to check that it is a "good" process and guide its construction and modification. For instance, one simple process completeness critic monitors the rule "the output of every task should be used by some other task." The same

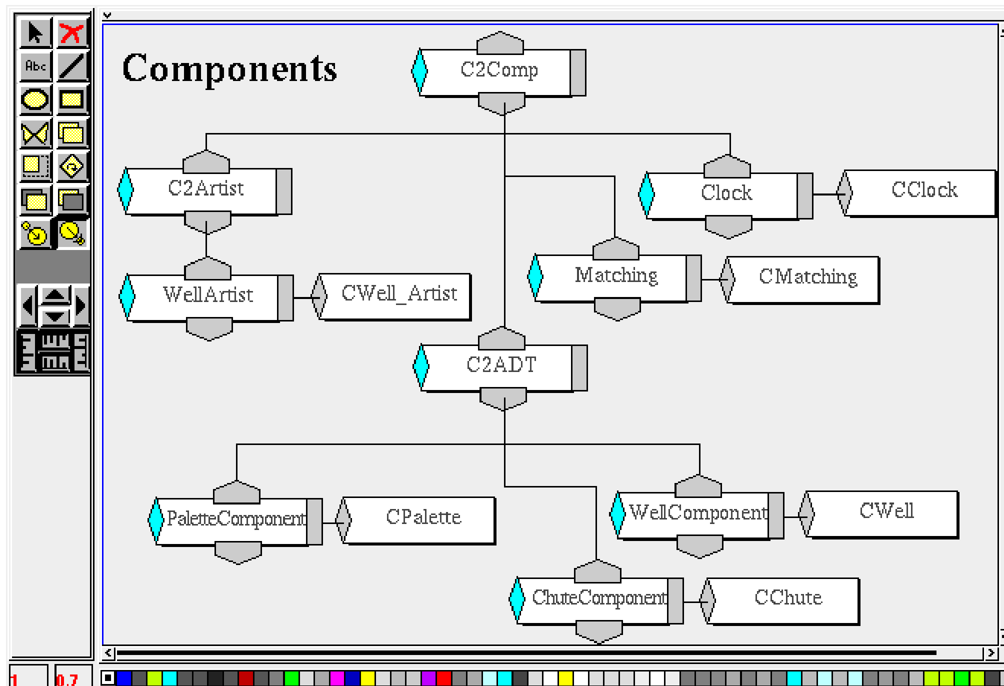

Figure 6: The KLAX Modular Perspective

techniques that are used to manage architecture critics can be used to manage process critics. Those techniques include modeling the meta-process and the architect's progress in it, so that process critics will be relevant and timely. While the ability to change the process gives freedom to individual architects, critics provide a mechanism to communicate or enforce externally imposed process constraints.

Support for Multiple Design Perspectives

Figure 2 shows a C2 conceptual architecture diagram produced with Argo. Small rectangles represent software components and connectors, arcs represent communication pathways. Small ovals on the components represent the communication ports of each component. The execution architecture hierarchically groups modules into operating system processes and threads (Figure 5). The modular architecture maps conceptual components to programming language modules (Figure 6). The source code storage architecture is not explicitly visualized, but annotations on modules give the pathnames of source files. The protocol perspective (not shown) presents type conformance relationships among interfaces of conceptual components. The modular and execution perspectives provide much of the information needed for system generation. The protocol and conceptual perspectives provide much of the information needed to check if communication requirements are being fulfilled. Information contained in any perspective can be used in analysis, simulation, or generation.

Argo allows multiple, coordinated perspectives with customization. In addition to the views described above, Argo allows for the rapid construction of new perspectives and their integration with existing perspectives. Simple perspectives may be programmed in a few hours by using a drawing editor to sketch the "look" of the perspective and then connecting architectural elements as appropriate. As noted in [21], architects which are given a fixed set of formal notations often revert to informal drawings when those tools are not applicable. One goal of Argo is to allow for the evolution of new notations as new needs are recognized. In addition to the structured graphics representing the architecture and process, Argo allows arbitrary, unstructured graphics to be drawn as annotations. Because the unifying structures of the system under construction must be communicated convincingly to other designers and system implementors, customizable presentation graphics are needed. Furthermore, ad-hoc annotations that are found to be useful can be incrementally formalized and incorporated into the notations of future styles. We expect that Argo's low barrier to customization will encourage evolution from unstructured notations to structured ones as recurring formalization needs are identified.

RELATED WORK

Significant research in design environment has been done at the University of Colorado at Boulder [6, 7, 8]. The human centered approach to design environments is emphasized in both Argo and the Colorado systems. Explicit specification of design goals is used in the Janus kitchen design environment, while checklists are present in the Framer application window layout design environment. However, Argo's "to do" list extends these in general by giving designers more control and customization. Argo adds prioritization of "to do" list items and allows items from sources other than critics. Overall, Argo gains strength from a tighter coupling to a first class design process model.

As noted in the introduction, the whole field of design environments relies on the conceptual framework of human augmentation pioneered by Engelbart [3]. His work provides a framework for a grand vision of human support, extended today to encompass cooperation of multiple human designers [4]. Although we situate our work within this field, we have focused our work to understand specific potential of human augmentation in software engineering. Restricting the scope further to software architecture design in a few application areas has allowed us to better link the augmentation mechanisms to the designer's situation. One example is that a newly initiated design session begins already with a close approximation to the user interface design process.

Several authors in the field of software architecture have identified the need for architectural design guidance [10, 24]. One approach to representing that knowledge is the compilation of architectural *styles*. The concept of styles is an approach to organizing design guidance, both constraining design decisions and factoring design complexity. When Argo is extended to support multiple architectural styles, it will be possible to organize critics and perspectives according to separate styles. Critics will continue to provide an active dimension to design guidance.

Soni, Nord, and Hofmeister [21] identify four architectural views: (1) Conceptual software architecture describes major design elements and their relationships. (2) Modular architecture describes the decomposition of the system into programming language modules. (3) Execution architecture describes the dynamic structure of the system. (4) Code architecture describes the way that source code and other artifacts are organized in the development environment. Their experience indicates that separating the concerns of each view leads to an overall architecture which is more understandable and reusable. In demonstrating Argo, we chose several of the same perspectives; however, we believe that the choice of perspectives depends on the application domain and the tasks and needs of design stakeholders.

Other software architecture design environments tend to emphasize automatic generation from formalisms over the cognitive needs of the architects. The Aesop [9] system allows for a style-specific design environment to be generated from a specification of the style. The DaTE [2] system allows for construction of a running system from a architectural description and a set of reusable software components. Argo can also generate main procedures which combine software components into a running system.

SUMMARY

The basic hypothesis that we have explored is that design environments for software architecture design, and design environments in general, need to support more than simply the artifacts involved: they must support the cognitive needs of the human designer.

The design of software architectures requires people to cope with myriad constraints imposed by implementation, architectural style rules, and organizational guidelines. Software architects have open to them a wide range of alternatives for most design decisions, and they require knowledge to evaluate those alternatives. Argo aids architects in navigating design decisions by providing critics the deliver relevant and timely knowledge.

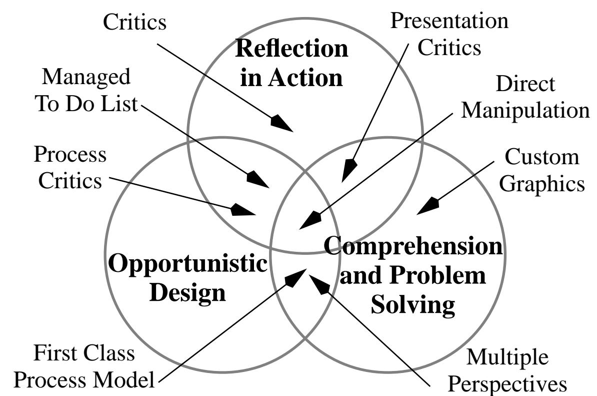

Figure 7: Overlapping Theories and Features

The process of designing a software architecture is complex and, in many cases, unpredictable, because designers follow processes opportunistically. Argo supports designers cognitive needs with a managed "to do" list, a simple process model that can be visualized, manipulated, and critiqued.

Human designers often cannot make the "right" design decisions if they cannot see the problem and their partial solution in the "right" way. Diverse design decisions require diverse visualizations. Argo provides for the definition of multiple, coordinated perspectives for interacting with designs.

For purposes of exposition, we have presented a direct mapping of cognitive theories to distinct design environment features. However, these features interact and overlap as suggested in our usage scenario and illustrated in Figure 7. The cognitive features do not determine the exact features needed in a given application domain. However, they do explain cognitive issues of design in a way that inspires, organizes, and motivates design environments.

ACKNOWLEDGMENTS

The authors would like to acknowledge Richard Taylor (UCI), Gerhard Fischer (CU Boulder), David Morley (Rockwell), and Peyman Oreizy, Nenad Medvidovic, and other members of the Chiron research team at UCI. We also thank the CSS'96 reviewers for their comments that significantly improved this paper.

REFERENCES

1. Abowd, G., Allen, R., and Garlan, D. Using style to understand descriptions of software architecture. *SIGSOFT Software Engineering Notes*, Dec. 1993, vol.18, (no.5), 9-20.
2. Batory, D. and O'Malley, S. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, Oct. 1992, vol.1, (no.4), 355-98.
3. Engelbart, D. A Conceptual Framework for the Augmentation of Man's Intellect. In: Greif I, ed. *Computer-*

Supported Cooperative Work: A Book of Readings. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1988, 35-66.

4. Engelbart, D. Toward Augmenting the Human Intellect and Boosting our Collective IQ. *Communications of the ACM* 1995; vol. 38, no. 8, 30-33.
5. Fischer, G. Cognitive View of Reuse and Redesign. *IEEE Software*, Special Issue on Reusability 1987; vol. 4, no. 4, 60-72.
6. Fischer, G., Girgensohn, A., Nakakoji, K., and Redmiles, D. Supporting software designers with integrated domain-oriented design environments. *IEEE Transactions on Software Engineering*, June 1992, vol.18, no.6, 511-22.
7. Fischer, G., Lemke, A., Mastaglio, T., and Morch, A. The role of critiquing in cooperative problem solving. *ACM Transactions on Information Systems*, April 1991, vol.9, no.2, 123-51.
8. Fischer, G., Lemke, A., McCall, R., and Morch, A. Making argumentation serve design. *Human-Computer Interactions*, 1991, vol.6, no.3-4, 393-419.
9. Garlan, D., Allen, R., and Ockerbloom, J. Exploiting style in Architectural Design Environments. *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1994. Software Engineering Notes, December 1994, vol 19, no.5, 175-88.
10. Garlan, D., Allen, R., and Ockerbloom, J. Architectural Mismatch: or Why it's hard to build systems out of existing parts. *International Conference on Software Engineering 17*, 1995, 179-185.
11. Guindon, R., Krasner, H., and Curtis, W. Breakdown and Processes During Early Activities of Software Design by Professionals. In: G.M. Olson ES S. Sheppard, ed. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing Corporation, Lawrence Erlbaum Associates, 1987, 65-82.
12. Kintsch, W. and Greeno, J. G. Understanding and Solving Word Arithmetic Problems. *Psychological Review* 1985;92, 109-129.
13. Medvidovic, N., Taylor, R. N., and Whitehead, Jr., E. J. Formal Modeling of Software Architectures at Multiple levels of Abstraction. *Proceedings of the California Software Symposium (CSS'96)*, Los Angeles, CA, USA, April 1996.
14. Pennington, N. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, Vol. 19, No. 1987, 295-341.
15. Perry, D. E. and Wolf A. L. Foundations for the Study of Software Architecture. *Software Engineering Notes*, October 1992, 40-52.
16. Polanyi, M. *The Tacit Dimension*.Garden City, NY: Doubleday, 1966.
17. Redmiles, D. Reducing the Variability of Programmers' Performance Through Explained Examples. *INTERCHI '93 Conference Proceedings*, April 1993, 67-73.
18. Rist, R. Variability in program design: the interaction of knowledge and process. *The International Journal of Man-Machine Studies* 1990, 1-72.
19. Schoen, D. *The Reflective Practitioner: How Professionals Think in Action*. New York: Basic Books, 1983.
20. Schoen, D. Designing as reflective conversation with the materials of a design situation. *Knowledge-Based Systems* 1992;vol. 5, no. 1, 3-14.
21. Soni, D., Nord, R., and Hofmeister C. Software Architecture in Industrial Applications. *International Conference on Software Engineering 17*, 1995, 196-207.
22. Soloway, E. and Ehrlich, K. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* 1984; vol. 10, no.5, 595-609.
23. Soloway, E., Pinto, J., Letovsky, S., Littman, D., and Lampert, R. Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM* 1988; vol. 31, no. 11, 1259-1267.
24. Taylor, R. N., Medvidovic, N., Anderson, K., Whitehead, Jr., E. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A Component and Message-based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, to appear.
25. Visser, W. More or less following a plan during design: opportunistic deviations in specification. *International Journal of Man-Machine Studies* 1990; 247-278.