

Software Architecture Design From the Perspective of Human Cognitive Needs

Jason E. Robbins and David F. Redmiles

Information & Computer Science
University of California, Irvine
Irvine, CA
{jrobbins, redmiles}@ics.uci.edu

Abstract

Much attention in software engineering research today is focussed on the notion of software architectures. The major motivation is that software architectures provide the appropriate level of abstraction to support the design of complex systems. The research has quickly evolved to the degree that design environments have been implemented to support software architects in creating new designs by combining components within architectural styles. We follow the same motivation with a different focus. We report on a software architecture design environment called Argo. Argo differs from other approaches by being paying attention to the human, cognitive needs software architects have during design as much as the representation and manipulation of the architecture itself. We emphasize the primary considerations by contrasting an analysis of the human, cognitive design process with a systems, software design process. The corresponding, key elements are illustrated through a design scenario with Argo. 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.

1: Introduction

Much attention in software engineering research today is focused on the use of software architectures in design [PW92, AAG93, GS93, TMA+95, BO92]. The major motivation is that software architectures provide the appropriate level of abstraction to support the design of complex systems. Furthermore, paralleling the research in software

reuse, the research in software architectures has the potential of yielding high quality products with the minimum effort. The research has quickly evolved to the degree that design environments have been implemented to support software architects in creating new designs by combining components within architectural styles. For example, Garlan and colleagues (as well as others) have explored the system requirements for supporting the definition, storage, retrieval, and application of architectural styles [GAO94]. They have also explored how these design environments may be adapted to a variety of domains.

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 [Engel88]. We have built a software architecture design environment, called Argo¹, 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.

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 critical feedback. The theory of opportunistic design has implications for how process representations should be used in a design environment. Finally, a theory of comprehension

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.

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 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.

2: 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 task? In what ways does the designer proceed to work through a design problem? 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.

The reader is reminded that these are not prescriptive theories (how or what design should be) but rather, they are descriptive theories (how and what people involved in design tasks have been observed doing). Hence the 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 (as described in Sections 3 and 4) are included to address these issues.

2.1: Reflection-in-Action

Donald Schön coined the term “reflection-in-action” to describe the behavior he observed of building architects working on new designs [Scho83]. 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 reframe the proposed, problem solution. Schön later referred to the process as a “conversation with the materials of design [Scho92].” This theory of design is further substantiated by the work of Polanyi’s on *tacit knowledge* [Pola66]. The theory demonstrated by Polanyi is that peoples’ abili-

ty 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 [GKC82]. Calling it “serendipitous design,” they noted that as the developers worked with the design, their mental model of the problem situation improved, and hence their design. Fischer characterizes this process as one of *co-evolution* [FGNR92]. 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, within his work, Schön 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 engineers 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 [FLMM91a]. A critic in this context is a routine that monitors a human designer’s progress in refining a new design artifact. The critic reacts to violations of some rule. The critic rule can encode a hard constraint that may not be overridden or it may encode a soft constraint that represents a suggestion. Critics are explored in more detail in Sections 3.1 and 4.1. Their general role is to allow the observed design process of reflection-in-action but to maintain positive properties of a prescribed software process. In particular, they augment a human designer’s ability to both detect potential breakdowns, especially in the situations where designers are working with unfamiliar components. Thus they impose some order on an otherwise unconstrained design process. Critics employ knowledge accumulated over past designs or by more experienced designers.

2.2: Opportunistic Design

Opportunistic design is a theory that developed out of empirical observations of mechanical and software engineers [Viss90, GKC82]. 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 [Viss90]. The theory of opportunistic design builds on the cognitive research by Soloway and others on programming plans [SE84] [SPLLL88]. In this context, programming plans represent the knowledge programmers have 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 [Rist90].

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; and 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 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 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, it is used 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 helps designers understand their progress and set priorities. Furthermore, designers may edit the process perspective and checklist, tailoring it to their current situation. Several other issues are discussed in Sections 3.2 and 4.2.

2.3: Comprehension and Problem Solving

Based on studies of people solving word algebra problems, Kintsch and Greeno developed a theory of problem solving behavior [KG85] and later Fischer and Kintsch extended the theory to explain the behavior of software engi-

neers [Fisc87]. 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. 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 [Red93]. In terms of the theory of opportunistic design, the hypothesis was to fill in designers' missing design schemas, reducing the cognitive cost of following a plan. An empirical study demonstrated that designers' understanding could be augmented effectively by explaining design components in multiple perspectives.

In Argo, multiple perspectives are again applied. These perspectives include conceptual component communication, component-to-module mapping, message protocol subsumption, 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.

3: Components for Software Architecture Design Environments

A domain oriented design environment [FGNR92] 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. The architecture is stored in an object model representing connected graphs. The designer views and manip-

ulates that representation through various perspectives such as component communication and implementation. Automated design critics in the environment monitor the design and deliver knowledge to the designer when relevance predicates are satisfied. The designer uses a process model as a resource in carrying out his design process. 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

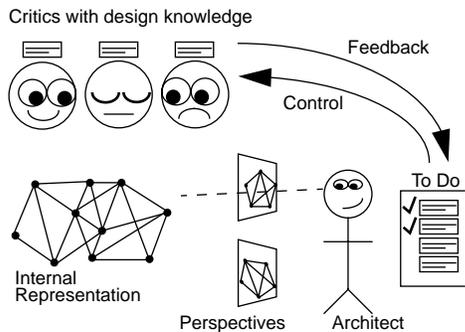


Fig. 1: Selected Argo components

3.1: 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 the most decisions are made. Design environments take a middle road, they critique the users design as it is being entered. Critical feedback is managed to inform the architect with minimal distraction. In the vast majority of cases critics simply advise the architect of potential errors or areas of improvement in the design, only the most severe errors are prevented outright.

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 interesting knowledge about software architectures will take the form of guidelines or rules of thumb.

Practical architectural styles will contain so many advisory rules that a non-disruptive feedback mechanism is needed.

There are a variety of potential types of critics. Each type of critic delivers design knowledge of a given type. 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 some design parameters.

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 in Section 2.1. Because critics are active from the beginning of the design process they can advise the architect as design decisions are explored. Organizing design analysis into critics imposes the burden on the critic author that partial analysis must be possible on a partial design, but yield the advantage that more information is available earlier. To reduce the burden on the critic author design environments manage critics and critical feedback so that overly pessimistic critics can inform without distracting the architects attention.

3.2: Cognitive Design Process

Each of the three cognitive theories discussed in Section 2 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 in Section 2.2. Designers deviate from plans, even their own plans. These deviations may be unavoidable or even sometimes desirable from a cognitive perspective, but lead designers into a variety of difficulties as discussed in the Guindon, Krasner, and Curtis study.

We respond to the observations on opportunistic design, and especially plan deviation, by maintaining a representation of the designers' task. The representation is based on representations for a software process, but is extended and applied in ways to accommodate the observed cognitive processes. Because of these extensions, we think of our use of process as adhering to a cognitive design process.

The foremost 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 as a resource.

Another characteristic is visibility. 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 to continue the design. Such excursions can lead to increased understanding of the design situation. Visibility of the process underlies the architects ability to modify based on their increased understanding.

Reminding is a critical aspect when designs are complex. Reminding helps architects revisit incomplete details or overlooked alternatives.

Flexibility, visibility, and reminding neatly work together to support delaying of detail design. If architects are forced to commit to each design decision that is begun before any analysis can be done, then the effort required explore design alternatives will be much higher. Higher effort means that fewer alternatives will be considered which reduce confidence in the design, and higher effort also shifts effort away from the design at hand and into planning required to use the tool efficiently. Furthermore, the ability of critics to deliver partial evaluation of partial designs in a managed and usable format allows designers to avoid 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, those decisions are made as part of a design process. For the critics to produce timely information they must have an explicit model of the design task and the architect's current status in that process. 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 it involves design decisions that the architect has not yet begun considering. The process model can also indicate when the architect considers a task finished, and the design environment can generate criticism at that time, perhaps marking the task as still in progress if there are high priority critical feedback items pending.

3.3: Multiple Design Perspectives

The architecture of a complex software system is itself a complex artifact. As that artifact is constructed it must be understood many times 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, separation of concerns into separate perspectives allows information relevant to certain related issues to be presented together and independently of information irrelevant to those issues.

In any complex structure 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, inter-component communication patterns, and naming conventions. Complex software systems by definition will not have a single unifying structure.

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 arises in the design, it may be appropriate to use a new perspective on the design in addressing those issues.

4: Description of Argo

This section describes the Argo software architecture design environment in detail. Features that support critical feedback, the human cognitive design process, and viewing the design from multiple perspectives are discussed in the following subsections.

Software architects use Argo by placing graphical representations of components on a drawing area and connecting them to make graph structures representing each of several perspectives of an architecture (Figures 4-6). Automated critics in the design environment deliver knowledge to the architects to aid them in making design decisions. The architects rely on the design feedback when they 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.

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 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 2). Architects may address issues at their own initiative.

The to-do list helps orient architects in the design process by enumerating possible issues that can be addressed next. Also supporting the architects' cognitive processes is a simple process model that describes steps for using Argo to make a new design (Figure 3). This model visually depicts the progress made at a given point in time. That simple process model and its presentation aids the architect in returning from "excursions in the design space."

Each architectural perspective is presented with a notation appropriate to that perspective. All perspectives can be presented by a connected graph where the nodes are the architectural elements of that perspective and the arcs are relationships among those elements. Each node in the graph is annotated with the attributes of one architectural element. Architectural elements which are relevant to multiple perspectives are presented once in each of those perspectives in a form that is appropriate to each. Argo allows for direct manipulation of connected graphs and annotations on the arcs and nodes. It is this simple, precise, and flexible design representation that allows Argo to work with diverse architectural perspectives.

The example used in the subsections that follow is further explained in [TMA+96]. The architectural style is that of *Chiron-2* [TMA+95], developed specifically for use in the domain of user interface software. Argo has been designed with the *Chiron-2* style in mind, although support for other styles is possible.

4.1: Support for Critics

Critics deliver knowledge to software architects about the implication of, or alternatives to a design decision. Critics are active objects which watch for specific conditions in the partial architecture as it is being constructed and notify the architects when those conditions are detected. Critics support a design process of action follow by re-

flection. The architects' abilities to reflect on a design is enhanced by the knowledge encoded in the critics.

Argo supports critics in two ways. First, it has a framework for implementing critics and a run-time facility for evaluating the predicates of active critics. Second, it has a variety of user interface mechanisms for controlling which critics are active at a given time and for managing critical feedback. Table 1 gives a description of some example critics.

Argo associates critics 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.

Feedback management techniques in Argo improve the capability of critics to trigger at the right time with the right information. First, critics may be active or inactive depending on the state of the architecture, the design process, and stated design goals. Second, the architects may *hush* critics, if the critics are providing inappropriate feedback or are otherwise determined to be too intrusive. Third, the architects choose when they will address an issue raised by a critic. Since critics insert their critiques into the ToDo list (see Figure 2), the architects may address issues in an order they choose and they may also reorder the list or insert items of their own. Fourth, the architects may explicitly solicit feedback through a "Critique" command. It gives every critic a chance to generate feedback.

4.2: Support for the Cognitive Design Process

As discussed in Section 2.2, 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 critical feedback is delivered only in a timely fashion.

TABLE 1. Some Argo Architectural Critics

Name of Critic	Knowledge Type	Relevant Design Task	Explanation
Invalid Connection	correctness	checking	Mandatory message signatures not satisfied by adjacent comps.
One Up One Down	correctness	checking	Violation of Chiron-2 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 below stated goals
OS Incompatibility	consistency	annotating	Components have conflicting environmental requirements

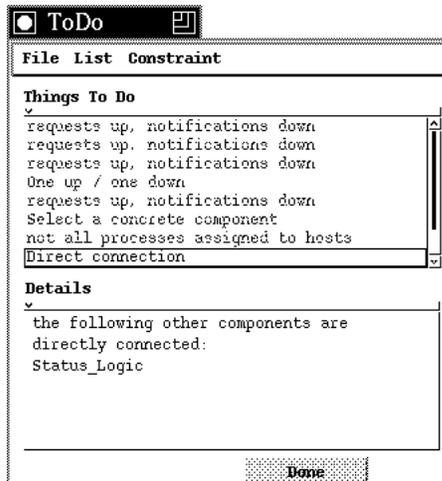


Fig. 2: The architects to do list

A model of the design process with status information provides leverage in managing critical feedback. We use an IDEF0-like notation to model the process tasks involved in a typical design process. (See Figure 3). Each

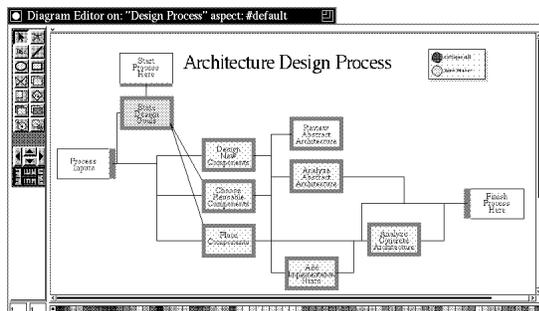


Fig. 3: Design process

task in the design process works on input produced by upstream tasks and produces output for consumption downstream tasks. No control model is mandated: tasks can be done in any order, 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 is also 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 process shown in Figure 3 is a fairly simple one, partly for purposes of presentation, partly because the Chiron-2 architectural style does not impose any 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 design process would be more complex if any of these limiting as-

sumptions were removed. Also, Argo allows multiple independent processes to be modeled simultaneously. For those reasons, the process of designing the process (usually called the meta-process) can itself be a complex, evolutionary task which could benefit from critical feedback.

The process model in Argo may be manipulated just as any other design artifact. Argo's infrastructure supports manipulation of any artifact in the general class of connected graphs. This support encompasses the critic mechanism as well. For instance, one simple process completeness critic monitors the rule "the output of every step should be used by some other step".

The same techniques that are used to manage architectural critics, can be used to manage process critics, including defining a meta-process and the architect's progress in defining the (normal) process. While the ability to change the process gives freedom to individual architects, the fact that those processes are critiqued provides a possible mechanism for enforcement of externally imposed process constraints.

4.3: Support for Multiple Design Perspectives

In any complex structure 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, inter-component communication patterns, and naming conventions. Complex software systems by definition will not have a single unifying structure. Furthermore, architects must work together with other architects and stakeholders, each of whom has their own task and background. For those reason, we use multiple views on the architecture.

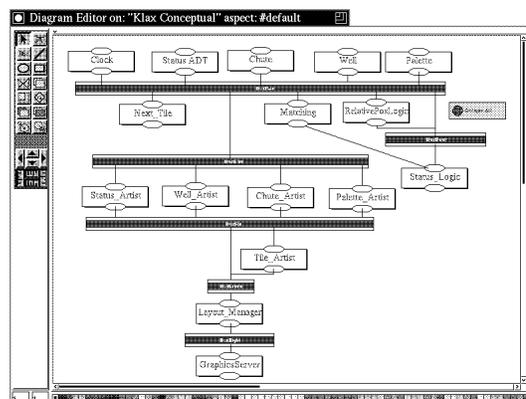


Fig. 4: The KLAX conceptual architecture

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 [GAO95, TMA+95]. One approach to representing that knowledge is the compilation of architectural styles. A software architecture 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 of system organization observed in a family of applications. Each style carries assumptions shared by components and connectors in that style, and assumptions about the domain in which the resulting systems will be used. When assumptions are carried in the style, the model of each component in that style becomes less complex. That is to say, architectural styles can be thought of as a technique for factoring complexity. We intend to use styles to organize critics and perspectives.

Soni, Nord, and Hofmeister [SNH95] identify four architectural views: Conceptual software architecture describes major design elements and their relationships. Modular architecture describes the mapping from conceptual components to programming language modules. Execution architecture describes the dynamic structure of the system. 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 improved reuse.

AnswerGarden [AM90] is a system which associates a domain expert with each part of a large organizational memory. Critic ownership attempts to make use of this same principle of involving the people who are in a position to change organizational knowledge or policy, and giving them the information that they need in order to make a decision. Bridget [GRS94] exploits the same principle in using expectation agents to allow end users to give feedback to user interface designers.

Other software architecture design environments have been developed. These systems tend to emphasize architectural styles and formalisms rather than user interface techniques. The Aesop [GAO94] system allows for a style-specific design environment to be generated from a specification of the style. The DaTE [BO92] system allows for construction of a running system from a architectural description and a set of reusable software components. Argo does not yet handle multiple architectural styles, but we anticipate that will be possible because of the fact that the critics responsible for evaluating style rules are stored with the descriptions of the reusable software components, not centrally to Argo itself. Argo can also generate main procedures which combine software components into a running system.

6: Conclusions

Software architectures are the unifying structures of software systems. The definition of software architecture is based on human characteristics. Architectures are the “big picture” of a software system, and as such they improve understanding of the system by the designers, implementors, users, and maintenance programmers. In searching for an architectural approach to software development we must keep the human cognitive needs central to our choice of paradigms, languages, and tools.

As in any design of complex artifacts, the software architect’s task demands knowledge to evaluate alternatives. Software architectures must satisfy 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. This paper has described the Argo design environment which aids architects in navigating design decisions by delivering knowledge in a timely manner.

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

Acknowledgments.

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.

The authors would like to acknowledge Richard Taylor (UCI), Gerhard Fischer (CU Boulder), David Morley (Rockwell), and Peyman Oreizy and other members of the Chiron research team at UCI.

References.

- [AAG93] G. Abowd, R. Allen, D. Garlan. Using style to understand descriptions of software architecture. SIGSOFT Software Engineering Notes, Dec. 1993, vol.18, (no.5):9-20.
- [AM90] M. Ackerman, T. Malone. Answer Garden: a tool for growing organizational memory. 1990 Conference on Office Information Systems. SIGOIS Bulletin, 1990, vol.11, (no.2-3):31-9.
- [BO92] D. Batory, S. O’Malley. 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.

- [Engel88] D. Engelbart. 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.
- [Engel95] D. Engelbart. Toward Augmenting the Human Intellect and Boosting our Collective IQ. *Communications of the ACM* 1995;38(8):30-33.
- [Fisc87] G. Fischer. Cognitive View of Reuse and Redesign. *IEEE Software*, Special Issue on Reusability 1987;4(4):60-72.
- [FGNR92] G. Fischer, A. Girgensohn, K. Nakakoji, D. Redmiles. Supporting software designers with integrated domain-oriented design environments. *IEEE Transactions on Software Engineering*, June 1992, vol.18, (no.6):511-22.
- [FLMM91a] G. Fischer, A. Lemke, T. Mastaglio, A. Morch. The role of critiquing in cooperative problem solving. *ACM Transactions on Information Systems*, April 1991, vol.9, (no.2):123-51.
- [FLMM91b] G. Fischer, A. Lemke, R. McCall, A. Morch. Making argumentation serve design. *Human-Computer Interactions*, 1991, vol.6, (no.3-4):393-419.
- [GAO94] D. Garlan, R. Allen, J. Ockerbloom. 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.
- [GAO95] D. Garlan, R. Allen, J. Ockerbloom. Architectural Mismatch: or Why it's hard to build systems out of existing parts. *International Conference on Software Engineering 17*, 1995. p. 179-185.
- [GKC87] R. Guindon, H. Krasner, W. Curtis. 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.
- [GRS94] A. Girgensohn, D. Redmiles, F. Shipman. Agent-Based Support for Communication between Developers and Users in Software Design. *Proceedings of the 9th Annual Knowledge-Based Software Engineering Conference*, 1994:22-29.
- [KG85] W. Kintsch, J.G. Greeno. Understanding and Solving Word Arithmetic Problems. *Psychological Review* 1985;92:109-129.
- [Pola66] M. Polanyi. *The Tacit Dimension*. Garden City, NY: Doubleday, 1966.
- [PW92] D. E. Perry, A. L. Wolf. Foundations for the Study of Software Architecture. *Software Engineering Notes*, October 1992. p.40-52.
- [Red93] D. Redmiles. Reducing the Variability of Programmers' Performance Through Explained Examples. *INTERCHI '93 Conference Proceedings*, April 1993:67-73.
- [Rist90] R. Rist. Variability in program design: the interaction of knowledge and process. *The International Journal of Man-Machine Studies* 1990;5:1-72.
- [Scho83] D. Schoen. *The Reflective Practitioner: How Professionals Think in Action*. New York: Basic Books, 1983.
- [Scho92] D. Schoen. Designing as reflective conversation with the materials of a design situation. *Knowledge-Based Systems* 1992;5(1):3-14.
- [SE84] E. Soloway, K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* 1984;SE-10(5):595-609.
- [SNH95] D. Soni, R. Nord, C. Hofmeister. Software Architecture in Industrial Applications. *International Conference on Software Engineering 17*, 1995. p. 196-207.
- [SPLLL88] E. Soloway, J. Pinto, S. Letovsky, D. Littman, R. Lampert. Designing Documentation to Compensate for Delocalized Plans. *Communications of the ACM* 1988;31(11):1259-1267.
- [TMA+95] R. N. Taylor, N. Medvidovic, K. Anderson, E. J. Whitehead, J. E. Robbins. A Component and Message-based Architectural Style for GUI Software. *International Conference on Software Engineering 17*, 1995. p. 295-304.
- [Viss90] W. Visser. More or less following a plan during design: opportunistic deviations in specification. *International Journal of Man-Machine Studies* 1990;33:247-278.