Extending Design Environments to Software Architecture Design

JASON E. ROBBINS, DAVID M. HILBERT, DAVID F. REDMILES

{jrobbins,dhilbert,redmiles}@ics.uci.edu

University of California, Irvine

Abstract. Designing a complex software system is a cognitively challenging task; thus, designers need cognitive support to create good designs. Domain-oriented design environments are cooperative problem-solving systems that support designers in complex design tasks. In this paper we present the architecture and facilities of Argo, a domain-oriented design environment for software architecture. Argo's own architecture is motivated by the desire to achieve reuse and extensibility of the design environment. It separates domain-neutral code from domain-oriented code, which is distributed among active design materials as opposed to being centralized in the design environment. Argo's facilities are motivated by the observed cognitive needs of designers. These facilities extend previous work in design environments by enhancing support for *reflection-in-action*, and adding new support for *opportunistic design* and *comprehension and problem solving*.

Keywords: Domain-oriented design environments, software architecture, human-computer interaction, human cognitive skills, evolutionary design

1. Introduction

Designing a complex software system is a cognitively challenging task; thus, designers need cognitive support to create good designs. Domain-oriented design environments (DODEs) are cooperative problem-solving systems that support designers in complex design tasks (Fischer and Lemke, 1988; Fischer et al., 1992; Fischer, 1994; Rettig, 1993). They are domain-oriented in that important concepts and constructs of a particular domain are provided by the environment: this helps close the gap between designers' knowledge and the notation used by the environment. They are cooperative in that they take into account the complementary strengths and weaknesses of humans and computer systems: designers focus on tasks such as specifying and adjusting design goals, decomposing problems into subproblems, and maintaining conceptual integrity; while the system supports designers by providing external memory, hiding non-essential details, checking for inconsistencies or potential design flaws, and providing design guidance, analysis, and visualization capabilities.

Domain-oriented design environments have been recognized as complementary to more traditional approaches to knowledge-based software engineering (Fischer, 1994). In contrast to program synthesis approaches (Partsch and Steinbruggen,

1983), DODEs provide a more interactive, iterative model that places emphasis on the evolutionary nature of design and the cognitive needs of designers.

Like program synthesis approaches, existing software architecture design tools have tended to be coarse-grained and discrete in approach. Design decisions are entered into a formal representation. That representation is fed as input to analysis tools which produce output regarding properties of the representation. Then, architects interpret the output, relate it back to design decisions embodied in the representation, and prepare the design for another iteration. In sum, existing approaches require the architect to perform batches of modifications between evaluation opportunities. This design process is coarse-grained, operating on whole architectures as units. The cognitive process is correspondingly coarse-grained, dealing with batches of decisions instead of individual decisions.

In contrast to this coarse-grained, discrete approach, we pursue a more fine-grained and concurrent approach based on DODEs. In the DODE approach, active agents, known as critics, perform analysis on partial architectural representations *while* architects are considering individual design decisions and modifying the architecture. Analysis is concurrent with decision-making so that architects are not forced to suspend the architecture's evolution or batch their decisions in preparation for analysis. Design feedback from critics can be used by architects while they are considering design decisions. Moreover, design feedback is directly linked to elements of the architecture thereby assisting architects in applying the feedback to revise the design. As will be discussed in Section 5, this approach more directly supports the evolutionary nature of the architecture design process and the cognitive needs of software architects.

The design environment facilities explored by Fischer and colleagues have provided an essential basis for our work. However, in building the Argo software architecture design environment we have extended previous work in two important ways: design environment extensibility and additional support for designers' cognitive needs. First, our architecture demonstrates a shift from a large, knowledge-rich design environment that manipulates passive design materials to a smaller, knowledge-poor design environment infrastructure that allows the user to interact with active, knowledge-rich design materials. Specifically, critics are associated with design materials stored in active design documents rather than being centralized in the design environment. Second, we extend previous design environment facilities by enhancing support for *reflection-in-action*, and adding new support for cognitive needs identified in the theories of *opportunistic design* and *comprehension and problem solving*. Specifically, we add a flexible process model and "to do" list to explicitly support opportunistic design, and multiple, coordinated design perspectives to aid in comprehension and problem solving.

This paper is organized as follows. In Section 2 we discuss previous work in design environments and existing software architecture design tools and styles. Section 3 presents an overview of Argo. Sections 4 and 5 discuss the two main contributions of our work. Section 4 is devoted to Argo's implementation, internal representations, and architecture. Section 5 describes the cognitive theories that have motivated

the facilities of Argo, and how these facilities extend previous work. Section 4 sets the stage for Section 5. Each cognitive theory is mapped to specific supporting features of Argo's implementation. Section 6 discusses evaluation of Argo. Finally, Section 7 presents conclusions and future work.

2. Previous Work

2.1. Design Environments and Cognitive Theories

A *domain-oriented design environment* is a tool that augments a human designer's ability to design complex artifacts. The concept of human augmentation is based on the work of Engelbart and others who researched ways computers could enhance peoples' performance of intellectual tasks (Engelbart, 1988, 1995). Design environments address systems-oriented issues such as design representation, transformations on those representations (e.g., code generation), and application of analysis algorithms. However, they go beyond most tools in their support for the designer's cognitive needs by including features specifically intended to address those needs.

The cognitive theory of *reflection-in-action* observes that designers of complex systems do not conceive a design fully-formed (Schoen, 1983, 1992). Instead, they must construct a partial design, evaluate, reflect on, and revise it, until they are ready to extend it further. For example, a software architect usually cannot decide in advance that a certain component will use certain machine resources. That decision is usually made in the context of other decisions about inter-component communication and machine resource allocation. A similar process can be observed when modifying an existing design. This theory suggests that design environments must provide design feedback to support decision-making in the context of partial designs, i.e. *while* designs are being manipulated.

Design environments support reflection-in-action with critics that give design feedback. *Design critics* are agents that watch for specific conditions in the partial design as it is being constructed and notify the designer when those conditions are detected. Critics deliver knowledge to designers about the implications of, or alternatives to, a design decision. In the vast majority of cases, critics simply advise the designer of potential errors or needed improvements; only the most severe errors are prevented outright, thus allowing the designer to work through invalid intermediate design states. Designers need not know that any particular type of feedback is available or ask for it explicitly. Instead, they simply receive feedback as they manipulate the design.

Designers can benefit from domain knowledge when it is delivered to them via critics. Even experienced designers need knowledge support in complex domains or when working with new design materials. The "thin spread of application domain knowledge" has been identified as a general problem in software development (Curtis, Krasner, and Iscoe, 1988). Examples of domain knowledge that can be delivered by critics include well-formedness of the design, hard constraints on the design, rules of thumb about what makes a good design, industry standards, orga-
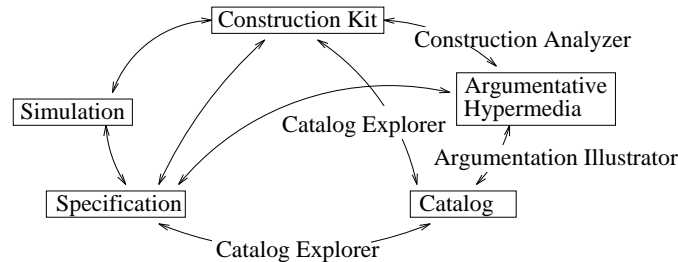
Figure 1. Design environment facilities of Janus, adapted from (Fischer, 1994).

nizational guidelines, and the opinions of fellow project stakeholders and domain experts.

Design environments such as Framer (Lemke and Fischer, 1990), Janus (Fischer, 1994), Hydra (Fischer et al., 1993), and VDDE (Voice Dialog Design Environment) (Sumner, Bonnardel, and Kallak, 1997) support reflection-in-action. Figure 1 shows facilities of this family of design environments. The domain-oriented construction kit facility allows users to visualize and manipulate a design. The construction analyzer facility critiques the design to give design feedback that is linked to hypertext argumentation. The goal specification facility helps to keep critics relevant to the designer's objectives. Reflection-in-action is also supported by simulation facilities that allow what-if analysis as a further design evaluation. A catalog of example designs can be accessed via the catalog explorer facility.

Designers will gain the most from design feedback that is both timely and relevant to their current design task. Design environments can address timeliness by linking critics to a model of the design process. For instance, Framer uses a checklist to model the process of designing a user interface window. At a given time the designer works on one checklist item and only critics relevant to that item are active. Design environments can address relevance by linking critics to specifications of design goals. For instance, Janus and Hydra allow the designer to specify goals for kitchen floorplans, and thus activate only those critics relevant to stated design goals. Furthermore, Hydra uses critiquing perspectives (i.e., explicit critiquing modes) to activate critics relevant to any given set of design issues and deactivate irrelevant critics.

2.2.   Architectural Styles

Work on software architecture (Perry and Wolf, 1992) has focused on representing systems as composed of software components and connectors (Garlan and Shaw, 1993). *Architectural styles* constrain and inform architectural design by defining the types of components and connectors available and the ways in which they may be combined (Abowd, Allen, and Garlan, 1993). Styles can be expressed as a set of

style rules. A simple architectural style is pipe-and-filter which defines components to be batch processes with standard input and output streams, and connectors to be data pipes. One pipe-and-filter style rule is that the architecture should contain no cycles.

Argo supports the *C2 architectural style* (Taylor et al., 1996). C2 is a component- and message-based style designed to model applications that have a graphical user interface. The style emphasizes reuse of UI components such as dialogs, structured graphics models, and constraint managers (Medvidovic, Oreizy, and Taylor, 1997).

The C2 style can be informally summarized as a layered network of concurrent components that communicate via message broadcast buses. Components may only send messages requesting operations upward, and notifications of state changes downward. Buses broadcast messages sent from one component to all components in the next higher or lower layer. Each component has a top and bottom interface. The top interface of a component specifies the notifications that it handles, and the requests it emits upward. The bottom interface of a component specifies the notifications that it emits downward, and the requests it handles.

2.3.   Software Architecture Design Tools

Design tools in the domain of software architecture have tended to emphasize analysis of well-formedness and code generation. The Aesop system allows for a style-specific design tool to be generated from a specification of the style (Garlan, Allen, and Ockerbloom, 1994). The DaTE system allows for construction of a running system from an architectural description and a set of reusable software components (Batory and O'Malley, 1992). Although not a software architecture tool, AMPHION is similar in that it allows users to enter a graphical specification from which the system can generate a running program (Lowry et al., 1994). Each of these systems provides support for design representation, manipulation, transformation, and analysis, but none of them explicitly supports architects' cognitive needs. Argo can generate code to combine software components into a running system; however, the main contribution of Argo to the software architecture community is its emphasis on cognitive needs.

KBSA/ADM (Benner, 1996) is a software design environment that embodies the results of many research projects stemming from a seminal vision of knowledge-based software development support (Green et al., 1983). KBSA/ADM has many features in common with Argo, including critics, a "to do" list, multiple coordinated models of the system under design, and process modeling. KBSA/ADM is intended to package previous research results into a full-featured software development environment. In contrast, Argo is intended to explore possible features that explicitly support identified cognitive needs. Support for cognitive needs in both KBSA/ADM and Argo is inspired by previous work in design environments, however we believe that Argo has a more integrated, reusable, and scalable infrastructure that yields better cognitive support.
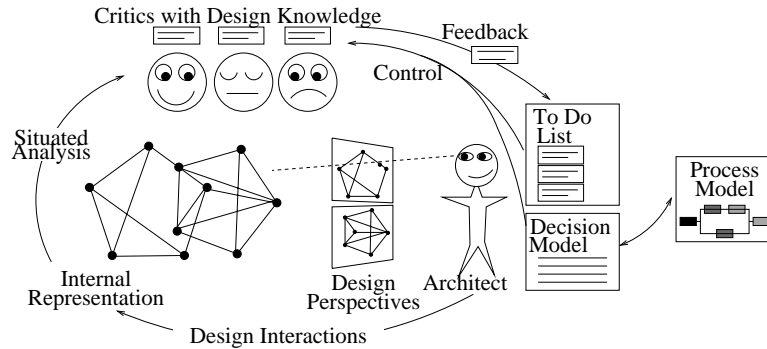
Figure 2. Design environment facilities of Argo.

3.   Overview of Argo

Figure 2 provides an overview of selected facilities of the Argo software architecture
design environment. The architect uses multiple, coordinated design perspectives
(Figure 3) to view and manipulate Argo's internal representation of the architecture
which is stored as an annotated, connected graph. Critics monitor the partial
architecture as it is manipulated, placing their feedback in the architect's "to do"
list (Figure 4). Argo's process model (Figure 5) serves the architect as a resource
in carrying out an architecture design process, while the decision model lists issues
that the architect is currently considering. Criticism control mechanisms use that
decision model to ensure the relevance and timeliness of feedback from critics.

   For comparison, Figure 1 shows facilities of the Janus family of design environ-
ments. Like Janus, Argo provides a diverse set of facilities to support reflection-in-
action including construction and critiquing mechanisms. Argo, however, extends
these facilities by integrating them with a flexible process model and "to do" list to
explicitly support opportunistic design, and multiple, coordinated design perspec-
tives to aid in comprehension and problem solving. Each of these cognitive theories
and the facilities that support them are discussed in Section 5.

   The subsections below describe each of Argo's facilities. The last subsection
provides a usage scenario that describes how an architect might interact with Argo.

3.1.   Critics

Critics support decision making by continuously and pessimistically analyzing par-
tial architectures and delivering design feedback. Each critic performs its analysis
independently of others, checking one predicate, and delivering one piece of design
feedback. Critics provide domain knowledge of a variety of types. *Correctness*
critics detect syntactic and semantic flaws. *Completeness* critics remind the archi-
tect of incomplete design tasks. *Consistency* critics point out contradictions within
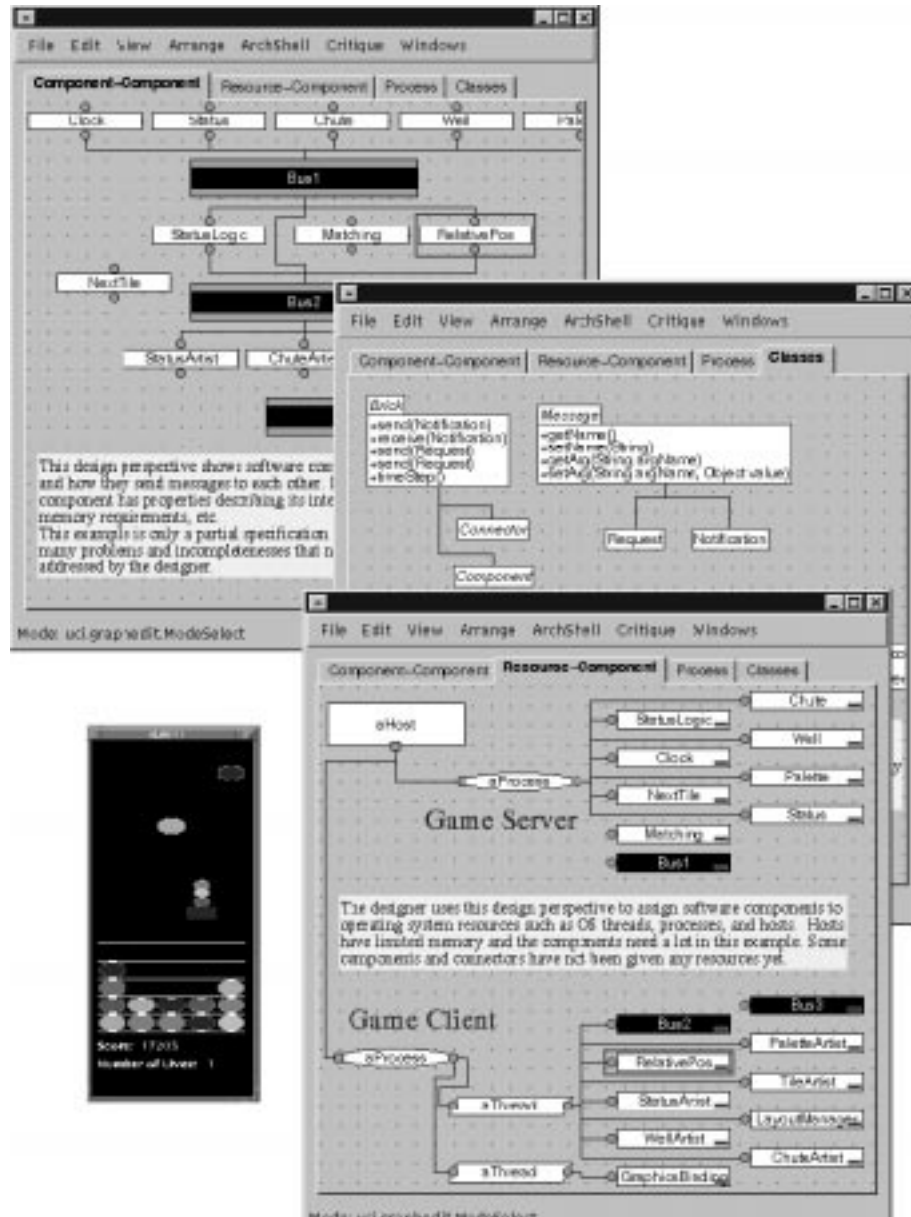
Figure 3. Three architecture design perspectives: the Component–Component perspective (top) shows conceptual component communication, the Classes perspective (middle) shows modular structure, and the Resource–Component perspective (bottom) shows machine and operating system resource allocation. The small window in the lower left shows the running KLAX game, represented by this architecture.

the design. *Optimization* critics suggest better values for design parameters. *Alternative* critics present the architect with alternatives to a given design decision. *Evolvability* critics consider issues, such as modularization, that affect the effort needed to change the design over time. *Presentation* critics look for awkward use of notation that reduces readability. *Tool* critics inform the architect of other available design tools at the times when those tools are useful. *Experiential* critics provide reminders of past experiences with similar designs or design elements. *Organizational* critics express the interests of other stakeholders in the development organization. These types serve to describe and aggregate critics so that they may be understood and controlled as groups. Some critics may be of multiple types, and new types may be defined, as appropriate, for a given application domain. Altogether, we have authored over fifty critics, including examples of each type. Some examples of architecture critics are given in Table 1.

We expect critics to be authored by project stakeholders for various reasons. An initial set of critics is developed by a domain engineer when constructing a domain-oriented design environment. Practicing architects may define critics to capture their experience in building systems and distribute those critics to other architects in their organization, or keep them for their own use in the future. A similar authoring activity was observed by Gantt and Nardi who found that groups of CAD tool users often had members they called "gardeners" that assumed the role of codifying solutions to local problems (Gantt and Nardi, 1992). Practicing architects may also refine existing critics by adding special cases to their predicates or by modifying their feedback. For example, one way for an architect to resolve criticism is to suggest a modification to the critic that raised the issue. Researchers may also define critics to support an architectural style. Existing literature on architectural styles and system design is a rich source of advice that can be made active via critics. Many organizations already have design guidelines that currently require designers to manually check their design. Software component vendors may define critics to add value to the components that they sell and to reduce support costs. For example, a critic supplied with an ASCII spell checking component might suggest upgrading to a Unicode version if the architect declares that internationalization is a goal.

Interactions among stakeholders in the design community can guide the evolution of critics. If the architect does not understand a particular critic's feedback or believes it to be incorrect, he or she may send structured email through Argo to the author of that critic. This opens a dialog between knowledge providers (i.e., domain experts) and consumers (i.e., practicing architects) so that the critics may be revised to be more relevant and timely. In this way critics can be thought of as *pro-active* "answers" in an organizational memory (Terveen, Selfridge, and Long, 1993; Ackerman and McDonald, 1996). Possible improvements to Argo's support for organizational memory include associating multiple experts with each critic, prioritizing experts based on organizational distance, and tracking email dialogs so that requests for changes are not forgotten.

Table 1. Selected Argo Architectural Critics

Name of Critic Critic Type	Decision Category	Explanation of Problem
Missing Memory Rqmts Completeness	Machine Resources	The memory required to run this component has not been specified.
Component Choice Alternative	Component Selection	There are other components that could "fit" in place of what you have: *list of components*.
Too Many Components Evolvability	Topology	There are too many components at the same level of decomposition to be easily understood.
Hard Combination to Test Organizational	Component Selection	If you need to use these components together, please make arrangements with the testing manager.
Generator Limitation Tool	Component Selection	The default code generator cannot make full use of this component.
Not Enough Reusable Components Consistency	Reuse	The fraction of components marked as being reusable is below your stated goals.
Avoid Overlapping Nodes Presentation	Readability	Overlapping nodes does not have any meaning in this notation and obscures node labels.
Portability Questionable Experiential	Portability	Your colleague, *name of person*, had difficulty using this component under *name of OS*.

3.2. Criticism Control Mechanisms

Formalizing the analyses and rules of thumb used by practicing software architects could produce hundreds of critics. To provide the architect with a usable amount of information, a subset of applicable critics must be selected for execution at any given time. Critics must be controlled so as to make efficient use of machine resources, but our primary focus is on effective interaction with the architect.

Criticism control mechanisms are predicates used to limit execution of critics to when they are relevant and timely to design decisions being considered by the architect. For example, critics related to readability should not be active when the architect is trying to concentrate on machine resource utilization. Computing relevance and timeliness separately from critic predicates allows critics to focus entirely on identifying problematic conditions in the product (i.e., the partial architecture) while leaving cognitive design process issues to the criticism control mechanisms. This separation of concerns also makes it possible to add value to existing critics by defining new control mechanisms.

3.3. The "To Do" List

Design feedback from large numbers of critics must be managed so as not to overwhelm or distract the architect. The *"to do" list* user interface (Figure 4) presents
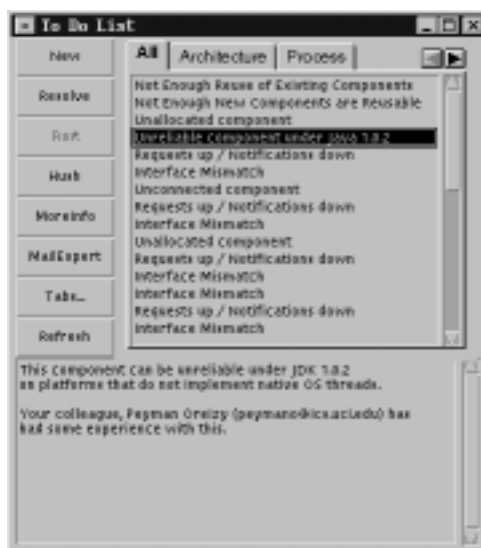
Figure 4. The architect's "to do" list.

design feedback to the architect in a non-disruptive way. When a "to do" item is added to the list, the architect may act on it immediately, or may continue manipulating the design uninterrupted. "To do" items come from several sources: critics post items presenting their analyses, the process model posts items to remind the architect to finish tasks that are in progress, and the architect may post items as reminders to return to deferred design tasks. Architects may address items in any order. Tabs on the "to do" list filter items into categories.

Each "to do" item is tied into the design context in which it was generated. That context includes the state of the design, background knowledge about the domain, and experts to contact within the design community. When the architect selects an item from the upper pane of the "To Do List" window, the lower pane displays details about the open design issue and possible resolutions. Double-clicking on an item highlights the associated (or "offending") architectural elements in all visible design perspectives. Once an item is selected, the architect may manipulate the critic that produced that item, send email to its author, or follow hyperlinks to background information.

3.4. Design Perspectives

A *design perspective* defines a projection (or subgraph) of the design materials and relationships that represent a software architecture. Perspectives are chosen to present only architectural elements relevant to a limited set of related design issues.

Figure 3 shows three perspectives on a system modeled in Argo. The system shown is a simple video game called KLAX[1] in which falling, colored tiles must be arranged in rows and columns. In the Component–Component perspective, nodes represent software components and connectors, while arcs represent communication pathways. Small circles on the components represent the communication ports of each component. The Resource–Component perspective hierarchically groups modules into operating system threads and processes. The Classes perspective maps conceptual components to classes in the hierarchy of programming language classes that implement them.

3.5. Process Model

Argo uses a process model to support architects in carrying out design processes. Design processes are difficult to state prescriptively because they are exploratory, tend to be driven by exceptions, and often change when new requirements, constraints, or opportunities are uncovered (Cugola et al., 1995). Rather than address traditional process modeling concerns (e.g., scheduling and enactment), our approach focuses on cognitive issues of the design process by annotating each task with the types of decisions that the architect must consider during that task. We use a simplified version of the IDEF0 process notation (IFIP, 1993) that models dependencies between tasks without prescribing a temporal ordering.

To support cognitive needs, Argo must maintain a model of some aspects of the architect's state of mind. Specifically, Argo's *decision model* lists decision types that the architect is currently considering. This information is used to control critics so that they are relevant and timely to the tasks at hand. The primary source of information used to determine the state of the decision model is decision type annotations on tasks in the process model. The architect may edit the decision model directly. Design manipulations performed by the architect can also indicate which decisions are currently being considered.

Figure 5 shows an example coarse-grained architecture design process model. Two of the tasks are to choose machine resources (`Choose Rsrcs`) and to choose reusable components (`Choose Comps`). The second task is annotated with the decision type `Reuse`. When the architect indicates that he or she is working on choosing reusable components, these annotations cause Argo to enable critics that support reuse decisions. The design process model shown in Figure 5 is a fairly simple one, partly because the C2 style does not impose any explicit process constraints, and partly because this example does not consider issues of organizational policy. In practice, the process would be more complex.

3.6. Usage Scenario

In this section we describe how an architect might interact with Argo while working through several steps in transforming the basic KLAX game (shown in Figure 3)
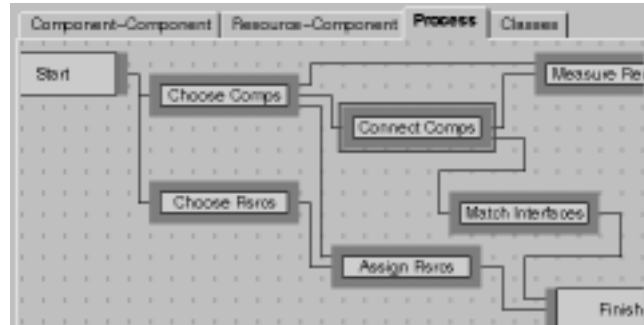
Figure 5. A model of the design process.

into a multi-player spelling game. The basic KLAX game uses sixteen separate components, including components that generate colored tiles, display those tiles, and determine when the player has aligned matching tiles. The spelling game variation will use the same basic architecture with new components to generate and display letters and to determine when the player has aligned letters to spell a word.

While working on the architecture of the basic KLAX game, the architect places the `TileArtist` component in the architecture. Shortly thereafter, an alternative critic posts a "to do" item indicating that another component from the company's library, `LetterArtist`, defines the same interface and should be considered as an alternative. The architect knows that `LetterArtist` is not appropriate for basic KLAX and takes no action, but the suggestion inspires the idea of building a spelling variation, so he or she leaves the item on the "to do" list. Later, when basic KLAX is completed, the architect reviews any remaining "to do" items and is reminded to investigate the spelling variation. He or she replaces `TileArtist` with `LetterArtist` and defines new components for `NextLetter` and `Spelling` to replace `NextTile` and `Matching`, respectively. While the architect is replacing these components the architecture is temporarily in an inconsistent state. Critics that check for consistency between component interfaces may post "to do" items describing these interface mismatches, but those items are automatically removed when the new components are connected and their interfaces are fully defined. Adding two new components to the architecture may cause a consistency critic to fire if the current percentage of reused components falls below stated reuse goals.

Satisfied with the choice of components and their communication relationships, the architect uses Argo's process model to decide what to do next. The process model contains a task for choosing reusable components and a task for allocating machine resources which depends on its output. At this point the architect decides to work on machine resource allocation and marks that task as *in progress*. Doing so enables critics that support design decisions related to machine resource allocation, and three new "to do" items are posted indicating that the three new components
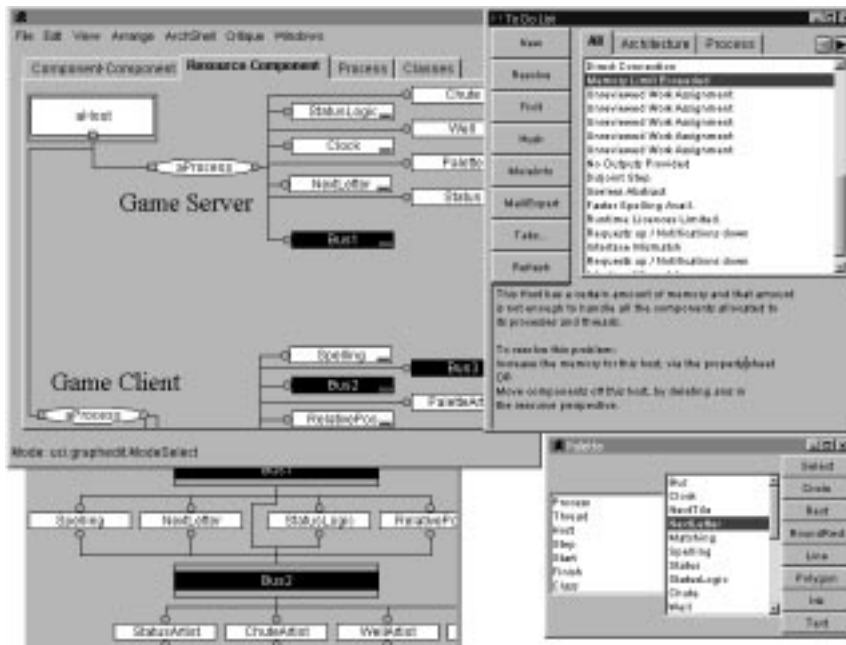
Figure 6. Architect's workspace after modifying KLAX.

have not been allocated to any host or operating system process. The architect then turns to the **Resource–Component** design perspective and finds that the nodes representing `TileArtist`, `NextTile`, and `Matching` have been removed and new nodes for `LetterArtist`, `NextLetter`, and `Spelling` have been added but not connected to any process or host. The architect connects the new components as the old ones were connected. At this point it occurs to the architect that a server-side `Spelling` component might be too slow in a future multi-player product, so he or she connects `Spelling` to the game client process instead. By viewing the **Component–Component** perspective and **Resource–Component** perspective the architect is able to understand interactions between two aspects of the architecture. Figure 6 shows what the architect would see at this point: two design perspectives are open and several new potential problems have been reported by critics. The selected "to do" item arose because the `Spelling` component requires more memory than is available.

In this usage scenario the architect has engaged in a constructive dialog with design critics: critics prompted the architect with new possibilities and pointed out inconsistencies. The architect used Argo's process model to help decide which design task to address next, and used two design perspectives to visualize and manipulate aspects of the architecture relevant to two distinct design issues. These

later two aspects of the scenario highlight new facilities of Argo that are not found in previous work on DODEs.

4.   Implementation

This section discusses the implementation of Argo. We base our discussion on two prototypes: an initial prototype implemented in Smalltalk and the current version implemented in Java. First we discuss the core elements of our approach: critics, criticism control mechanisms, perspectives, and processes. We then describe Argo's own architecture and the representation of architectures being designed with Argo.

4.1.   Critics

In Argo, a critic is implemented as a combination of (1) an analysis predicate, (2) type and decision category attributes for determining relevance, and (3) a "to do" list item to be given as design feedback. The stored "to do" list item contains a headline, a description of the issue at hand, contact information for the critic's author, and a hyperlink to more information. We encode critics as programming language predicates. Determining which languages are best suited for expressing critics is a topic for future research. Each critic is associated with one type of design material and is applied to all instances of that type. Critics may access the attributes of the design materials they are applied to, and traverse relationships to other design materials. Critic predicates are written *pessimistically*: unspecified design attributes are assumed to have values that cause the critic to fire. Table 2 presents one critic in detail.

Argo provides a critic run-time system that executes critics in a background thread of control. Critics may be run periodically or be triggered by specific architecture manipulations. During execution a critic applies its analysis predicate to evaluate the design and posts a copy of its "to do" item, if appropriate. Another thread of control periodically examines each item on the architect's "to do" list and removes items that are no longer applicable.

4.2.   Criticism Control Mechanisms

Criticism control mechanisms are implemented as predicates that determine if each critic should be enabled. Argo uses several criticism control mechanisms, any one of which can disable a critic. In each of the following examples, criticism control mechanisms decide which critics should be enabled by comparing information provided by the architect to attributes on the critics. Architects may "hush" individual critics, rendering them temporarily disabled, if their feedback is felt to be inappropriate or too intrusive. This allows architects to defer the issues raised by a particular critic without risk of leaving the critic disabled indefinitely. Argo's user

Table 2. Details of the Invalid Connection critic

Attribute	Value
Name	Invalid Connection
Design Material	`Component`
Types	{ `Correctness` }
Decision Categories	{ `Component Selection, Message Flows` }
Hushed	False

Smalltalk Predicate	`[:comp | | invalidServices |`
	`  invalidServices := comp inputs , comp outputs`
	`    select: [:s | s isSatisfied not].`
	`  invalidServices isEmpty not. ]`

Feedback	This component needs the following messages be sent or received, but they are not present: *a list of messages*
Author	`jrobbins@ics.uci.edu`
MoreInfo	`http://www.ics.uci.edu/pub/arch/argo/v05/docs/...`

interface allows groups of critics to be enabled or disabled by type. This allows the architect to control groups of critics easily. Another control mechanism checks the critic's decision types against those listed in the decision model. This keeps critics relevant to the tasks at hand.

Criticism control mechanisms normally enhance relevance and timeliness. However, relevance and timeliness can be reduced if criticism control mechanisms use incorrect information. For example, if the architect mistakenly indicates that a given issue is not of interest, then the architect will see no feedback related to that issue and might mistakenly assume that the architecture has no problems. This situation can be avoided by hushing critics instead of disabling them and by using a well designed process that reminds the architect to review all the issues. Argo advises the architect to check the decision model when the "to do" list becomes overly full or if too many "to do" items are being suppressed. The number of suppressed "to do" items is computed by occasionally running disabled critics without presenting their feedback.

4.3. Design Perspectives

In Argo, perspectives are objects that define a subgraph of the design materials in the current design. Two types of perspectives are defined in Argo: predicate and ad-hoc. *Predicate* perspectives contain a predicate that selects a subgraph of the design. *Ad-hoc* perspectives contain an explicit list of design materials and relationships. This latter mechanism allows for manual construction of perspectives via a diagram editor. When a new design material instance is added to the design, predicate

perspectives automatically include it if appropriate, whereas ad-hoc perspectives will only contain the new material if it is explicitly added to that perspective.

4.4. Process Model

Argo's process modeling plug-in provides a simplified process modeling notation based on IDEF0 (Figure 5). The design process is modeled as a task network, where each task in the design process works on input produced by upstream tasks and produces output for use by downstream tasks. No control model is mandated: tasks can be performed 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 *finished.* Each task is also marked with a list of decision types. Status information is shown graphically via color in the process diagram. These attributes are used to update the decision model. When the architect indicates that a task is considered finished, the design environment can use this cue to generate additional criticism, perhaps marking the task as still in progress if there are high priority "to do" items pending.

The process of defining and evolving the process (referred to as the meta-process) is itself a complex, evolutionary task for which architects may need support. The process model in Argo is first-class: it is represented as a connected graph of active design materials and the architect may define and modify the process model via the same facilities used to work on architectures. Multiple perspectives may be defined to view the process. Critics may operate on the process model to check that it is well-formed and guide its construction and modification, e.g., the output of this task should be used by another task. The same techniques used to control architecture critics can be used on process critics, including modeling the meta-process so that process critics will be relevant and timely. While the ability to change the process gives flexibility to individual architects, process critics can communicate or enforce external process constraints.

4.5. Design Environment Architecture

Figures 1 and 2 indicate what facilities are available to architects, but they give little indication of how the design environment is implemented. Janus and similar systems have tended to have one major software component for each facility. Those components form a knowledge-rich design environment with tight user interface, data, and control integration (Thomas and Nejmeh, 1992). Our interest in software architecture motivated us to seek a more flexible and extensible architecture, while retaining a fairly high level of integration.

Figure 7 presents Argo's architecture as a virtual machine. The lowest layer provides domain-neutral infrastructure and user interface components including support for representing connected graphs, multiple perspectives, the critic run-time system, "to do" list, and logging facilities. Domain-specific plug-ins are built

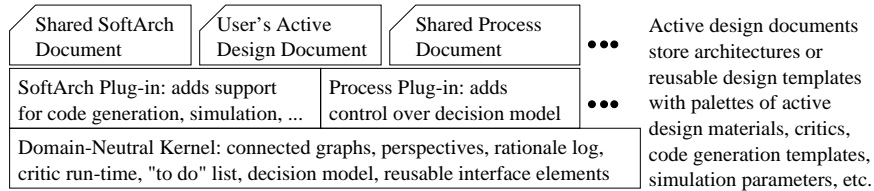| Shared SoftArch Document | User's Active Design Document | Shared Process Document | ●●● | Active design documents store architectures or reusable design templates with palettes of active design materials, critics, code generation templates, simulation parameters, etc. |
| SoftArch Plug-in: adds support for code generation, simulation, ... | Process Plug-in: adds control over decision model | | ●●● | |
| Domain-Neutral Kernel: connected graphs, perspectives, rationale log, critic run-time, "to do" list, decision model, reusable interface elements | | | | |

Figure 7. Argo's architecture presented as a virtual machine.

on top of that infrastructure if needed. These plug-ins typically provide pervasive functionality that cannot be built into any particular design material. For example, code generation support is useful for all design materials in the software architecture domain. Most domain-oriented artifacts are stored in "active documents" in the top layer. These documents are active in that they contain design materials (e.g., software components) that carry their own domain knowledge and behavior in the form of critics, simulation routines, and code generation templates. Documents may contain palettes of design materials, designs, reusable design templates, process fragments, or other supporting artifacts.

One advantage of this architecture is that artifacts from various supporting domains may be used. Here, a *domain* is a coherent body of concepts and relationships found in a given problem area, and a *supporting domain* is a domain for a problem area of secondary concern to the designer, but is useful in completing the design task. For example, a software architect's primary design domain is the construction of systems from software components, whereas recording and managing design rational is a domain of concern that is important to architects but secondary to construction. In Argo, plug-ins for software architecture, process modeling, and design rationale may all be available simultaneously, providing software architects with first-class supporting artifacts for process and rationale. Each supporting artifact may be manipulated, visualized, and critiqued.

In designing this architecture we shift away from a monolithic, knowledge-rich design environment that manipulates passive design materials to a modular, domain-neutral infrastructure that allows the architect to interact with knowledge-rich, active design materials. The same trend toward distributing knowledge and behavior to the objects of interest can be observed in the general rise of object-oriented and component-based approaches to software design. Active design materials can be thought of as first-class objects with local attributes and methods. The analysis predicates of critics can be thought of as methods. Critics that cannot easily be associated with any one design material may be associated with one or more design perspectives. For simplicity, Figure 2 presents critics as looking down on the design from above; a more literal presentation would show critics associated with each node, looking around at their neighbors.

The advantages of this shift include increased extensibility, scalability, and separation of concerns in the design environment, and stronger encapsulation of design

materials. Encapsulation is enhanced because attributes needed for analysis can be made local, or private, to the design materials, thus supporting local name spaces and data typing conventions. This increases extensibility because each design material may be packaged with its own analyses, and thus define its own semantics, which need not be anticipated by the design environment builder. Scalability in the number of critics is increased because there is no central repository of critics—critics simply travel with design materials. Concerns are separated because the design environment only provides infrastructure to support analyses packaged as critics and need not perform any analysis itself. All of these advantages support the evolution of architectures, design environments, and software architecture communities over time.

Effective support for diverse design decisions depends on the architect's ability to obtain and manage large numbers of critics. In the Java version of Argo, design materials and critics may be dynamically loaded over the Internet. For example, in a software component marketplace, an architect might download several component design materials, try them in the current architecture, consider the resulting design feedback, and make an informed component selection.

5.   Cognitive Theories and Extensions to the DODE Approach

Our extensions to previous design environment facilities are motivated by theories of designers' cognitive needs. Specifically, we extend previous design environment facilities by enhancing support for reflection-in-action and adding new support for cognitive needs identified in the theories of opportunistic design and comprehension and problem solving. These theories identify the cognitive needs of designers and serve to define requirements on design environments. In the subsections below we describe how Argo addresses these requirements. Table 3 summarizes Argo's support for cognitive needs.

5.1.   Reflection-In-Action

5.1.1.   Theory

As discussed in Section 2.1, Schoen's theory of *reflection-in-action* indicates that designers iteratively construct, reflect on, and revise each intermediate, partial design. Guindon, Krasner, and Curtis note the same effect as part of a study of software developers (Guindon, Krasner, and Curtis, 1987). Calling it "serendipitous design," they noted that as the developers worked hands-on with the design, their mental model of the problem situation improved, hence improving their design.

Software architectures are evolutionary artifacts in that they are constructed incrementally as the result of many interrelated design decisions made over extended periods of time. We visualize design as a process in which a path is traced through a space of branching design alternatives. A particular software architecture can be

Table 3. Argo features and the cognitive theories that they support.

	Critics	Low barriers to authorship	Continuous and pessimistic	Kept relevant and timely	Produce feedback with links	**To do list**	Presents feedback	Reminders to oneself	Allows choice	**Process model**	Process editing	Process critics	Process perspectives	**Design perspectives**	Multiple, overlapping	Customizable	Presentation critics
Reflection-in-action																	
Diversity of knowledge		X															
Evaluation during design		X	X														
Providing missing knowledge				X	X												
Opportunistic design																	
Timliness		X															
Reminding							X	X					X				
Process flexibility									X			X	X				
Process guidance							X					X					
Process visibility							X						X				
Comp. & problem solving																	
Dividing complexity															X		
Multiple perspectives that match multiple mental models															X	X	X

thought of as a product of one of the possible paths through this space. Choices at any point can critically affect alternatives available later, and every decision has the potential of requiring earlier decisions to be reconsidered.

Traditional approaches to software architecture analysis require architects to make numerous design decisions before feedback is provided. Such analyses evaluate the products of relatively complete paths through design space, without providing much guidance at individual decision points. As a result, substantial effort may be wasted building on poor decisions before feedback is available to indicate the existence of problems, and fewer design alternatives can be explored. Furthermore, when analysis is performed only after extended design episodes, it may be difficult to identify where exactly in the decision path the architect initially went wrong.

Diverse analyses are required to support architects in addressing diverse design issues, such as performance, security, fault-tolerance, and extensibility. Research to date has produced a diverse set of architectural analysis techniques. They include static techniques, such as determining deadlock based on communication protocols between components (Allen and Garlan, 1994) and checking consistency between architectural refinements (Moriconi, Qian, and Riemenschneider, 1995), as well as dynamic techniques, such as architecture simulation (Luckham and Vera, 1995).

The need for diversity in analysis is further driven by the diversity in project stakeholders and the potentially conflicting opinions of experts in the software architecture field itself (Garlan, 1995). Curtis, Krasner, and Iscoe note conflicting requirements (and thus design evaluation criteria) as a major problem for software development in general (Curtis, Krasner, and Iscoe, 1988). Conflict will naturally arise in architecture design, and analysis techniques should be capable of accommodating it. Accommodating conflict in analysis yields more complete support, whereas forbidding conflict essentially prevents architects from being presented with multiple sides of a design issue. Consider architectural styles, which provide design guidance by suggesting constraints on component and connector topology: a given architecture may satisfy the rules of several diverse styles simultaneously. Feedback items related to each of those styles can be useful, even if they contain conflicting advice.

5.1.2.   Support in Argo

Argo supports reflection-in-action with critics and the "to do" list. Critics deliver knowledge needed to evaluate design decisions. The "to do" list serves as a knowledge "in-box" by presenting knowledge from various sources. We will soon add visual indicators to draw the architect's attention to design materials with pending criticism (Silverman and Mezher, 1992; Terveen, Stolze, Hill, 1995). The "to do" list and informative assumption (described below) together support decision making by allowing the architect to browse potential design problems, guideline violations, and expert opinions.

Existing software analysis techniques are extremely powerful for detecting well-defined properties of completed systems, such as memory utilization and performance. These approaches adhere to what we call the *authoritative assumption*: they support architectural evaluation by proving the presence or absence of well-defined properties. This allows them to give definitive feedback to the architect, but may limit their application to late in the design process, after the architect has committed substantial effort building on unanalyzed decisions.

Such approaches also tend to use an interaction model that places a substantial cognitive burden on architects. For example, architects are usually required to know of the availability of analysis tools, recognize their relevance to particular design decisions, explicitly invoke them, and relate their output back to the architecture. This model of interaction draws the architect's attention away from immediate design goals and toward the steps required to get analytical feedback. Explicit

invocation of external tools scales well in terms of machine resources, but not in terms of human cognitive ability. We believe the cognitive burden of interacting with external tools may be enough to prevent their effective use.

Argo follows the DODE tradition in using what we call the *informative assumption*: architects are ultimately responsible for making design decisions, and analysis is used to support architects by informing them of potential problems and pending decisions. Critics are pessimistic: they need not go so far as to prove the presence of problems; in fact, formal proofs are often not possible, or even meaningful, on partial architectures.

Heuristic analyses can identify problems involving design details that may not be explicitly represented in the architecture, either because the model is too abstract, or because the architecture is only partially specified. Critics can pessimistically predict problems *before* they are evident in the partial design, and positively detect problems very quickly after they are evident in the partial design, typically within seconds of the design manipulation that introduces the problem. Criticism control mechanisms help trade early detection for relevance to current goals and concerns. In cases where all relevant design details are specified, critics can produce authoritative feedback.

Unfortunately, for most design issues, there are inherit trade-offs that prevent achieving both informative and authoritative feedback. There will always be a gap between the making of a decision and the analysis of that decision. That gap allows the passing of time, expenditure of effort, and loss of cognitive context. When one decision is analyzed in isolation, the gap may be small, but the feedback is at best informative because that decision interacts with others that have not yet been made. When analysis is deferred until groups of interrelated decision have all been made, the gap is necessarily larger, but the feedback may be more authoritative because more interactions are known.

However, there is a compromise for the informative–vs–authoritative tradeoff: existing analysis tools can be modified to make pessimistic assumptions in cases where partial architectures lack information needed for authoritative analysis; and existing critics can be controlled so as to achieve a certain degree of confidence before providing feedback. Alternatively, external batch analysis tools can be paired with *tool* critics that remind the architect when those tools would be useful. For example, a tool critic could watch for modifications that affect the result of the batch analysis and check that the architecture is in a state that can be analyzed (i.e., it has no syntax errors that would prevent that particular analysis), then re-run the batch tool, and parse its output into "to do" items with links back to the design context. In this case the critic's knowledge is of tools available in the development environment and when they are applicable, whereas the tools themselves provide architectural or domain knowledge.

Reusing existing analysis tools is one way to produce new critics, but we expect most critics to be written and modified by domain engineers, domain experts, vendors, or practicing architects. Argo's approach helps to ease critic authoring in that critics are pessimistic, critic authors need not coordinate their activities with

other authors to avoid giving conflicting advice, and critics need not consider relevance and timeliness. Argo's infrastructure eases critic authoring by providing a framework for implementing critics, a user interface for managing critics and their feedback, and templates for critics and their "More Info" web pages.

5.2.   Opportunistic Design

5.2.1.   Theory

It is customary to think of solutions to design problems in terms of a hierarchical plan. Hierarchical decomposition is a common strategy to cope with complex design situations. However, in practice, designers have been observed to perform tasks in an opportunistic order (Hayes-Roth and Hayes-Roth, 1979; Guindon, Krasner, and Curtis, 1987; Visser, 1990). The cognitive theory of *opportunistic design* explains that although designers plan and describe their work in an ordered, hierarchical fashion, in actuality, they choose successive tasks based on the criteria of cognitive cost. Simply stated, designers do not follow even their own plans in order, but choose steps that are mentally least expensive among alternatives.

The cognitive cost of a task depends on the background knowledge of designers, accessibility of pertinent information, and complexity of the task. Designers' background knowledge includes their design strategies or schemas (Soloway et al., 1988). If they are lacking knowledge about how to structure a solution or proceed with a particular task, they are likely to delay this task. Accessibility of information may also cause a deviation in planned order. If designers must search for information needed to complete a task, that task might be deferred. Complexity of a task roughly corresponds to the number of smaller tasks that comprise it.

Priority or importance of a step is the primary factor that supersedes the least cost criteria. Priority or importance may be set by external forces, e.g., an organizational goal or a contract. Designers may also set their own priorities. In some observations, designers placed a high priority on overlooked steps or errors (Visser, 1990).

Thus, the theory of opportunistic design outlines a "natural" design process in which designers choose their next steps to minimize cognitive cost. However, there are inherent dangers in this "natural" design process. Mental context switches occur when designers change from one task to another. When starting a new step or revisiting a former one, designers must recall schemas and information needed for the task that were not kept in mind during the immediately preceding task. Inconsistencies can evolve undetected. Some requirements may be overlooked or forgotten as the designer focuses on more engaging ones. Efficiency is lost because of many context switches. Guindon, Krasner, and Curtis observed the following difficulties.

> The main breakdowns observed are: (1) lack of specialized design schemas; (2) lack of a meta-schema about the design process leading to poor allocation of resources to the various design activities; (3) poor prioritization of issues

> leading to poor selection of alternative solutions; (4) difficulty in considering
> all the stated or inferred constraints in defining a solution; (5) difficulty in
> performing mental simulations with many steps or test cases; (6) difficulty
> in keeping track and returning to subproblems whose solution has been post-
> poned; and (7) difficulty in expanding or merging solutions from individual
> subproblems to form a complete solution. (Guindon, Krasner, and Curtis,
> 1987)

One implication is that designers would benefit from the use of process modeling.
Common process models support stakeholders in carrying out prescribed activities,
e.g., resolving a bug report. Software process research has focused on developing
process notations and enactment tools that help ensure repeatable execution of
prescribed processes. However, in their focus on repeatable processes, process tools
have tended to be restrictive in their enforcement of process steps.

Design environments can allow the benefits of both an opportunistic and a pre-
scribed design process. They should allow, and where possible augment, human
designers' abilities to choose the next design task to be performed. They can help
designers by providing information so they do not make a context switch. Process
support should exhibit the following characteristics to accommodate the real design
process as described by the theory of opportunistic design and address the problems
identified by Guindon, Krasner, and Curtis.

Visibility helps designers orient themselves in the process, thus supporting the
designer in following a prescribed process while indicating opportunities for choice.
The design process model should be able to represent what has been done so far
and what is possible to do next. Visibility enables designers to take a series of
excursions into the design space and re-orient themselves afterwards to continue
the design process.

Flexibility allows designers to deviate from a prescribed sequence and 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 improves. The process model should serve primarily as a resource
to designers' cognitive design processes and only secondarily as a constraint on
them. Allowing flexibility increases the need for guidance and reminding.

Guidance suggests which of the many possible tasks the designer should perform
next. Opportunistic design indicates that cognitive costs are lower when tasks are
ordered so as to minimize mental context switching. Guidance sensitive to priorities
(e.g., schedule constraints) must also be considered. Guidance can include simple
suggestions and criticisms. It may also include elaborate help, such as explanations
of potential design strategies or arguments about design alternatives.

Reminding helps designers revisit incomplete tasks or overlooked alternatives.
Reminding is most needed when design alternatives are many and when design
processes are complex or driven by exceptions.

Timeliness applies to the delivery of information to designers. If information
and design strategies can be provided to designers in a timely fashion, some plan
deviations and context switches may be avoided. Achieving timeliness depends on

anticipating designers' needs. Even an approximate representation of designers' planned steps can aid in achieving timeliness.

5.2.2. Support in Argo

Motivated by the theory of opportunistic design, we have attempted to move from predefined processes that force a certain ordering of design decisions to flexible process models with the properties outlined above. We extend previous work in design environments by introducing an explicit model of the design process with progress information and a more flexible "to do" list user interface for presenting design feedback.

Argo's process model supports visibility by displaying the process and the architect's progress in it. Visibility is further supported by the availability of multiple perspectives on the process. For example, an architect may choose a perspective that shows only parts of the process that lead to a certain goal. Furthermore, the "to do" list presents a list of issues that the architect may consider next.

Several authors have noted that traditional, sequential work-flow systems do not adequately support flexibility and proposed the use of constraint-based process models (Dourish et al., 1996; Glance, Pagani, and Pereschi, 1996). In Argo, flexibility is allowed by the simple fact that Argo does not use the process model to constrain the architect's actions: the architect may address any "to do" item or perform any design manipulation at any time. Furthermore, flexibility is *supported* by the architect's ability to modify the process model to better represent their mental model of the design process. Process critics, process perspectives, and a meta-process all support the architect in devising a good design in the process domain.

In the current version of Argo, guidance is provided only implicitly by the layout of the process model and the prioritization of the "to do" items. However, the theory of opportunistic design suggests that guidance should be based, in part, on the mental context required to perform each task. Pending "to do" items could be prioritized based on a rough estimate of the cognitive cost of addressing them.

The "to do" list and process model together support reminding by showing the issues that are yet to be addressed. The "to do" list reminds the architect of issues that can be addressed immediately while the process model shows tasks that must be addressed eventually. Critics and "to do" items remind the architect of issues that need to be reconsidered as problems arise. Beyond the knowledge contained in the critics and the process model, the architect can also create "to do" items that contain arbitrary text and links as personal reminders.

The continuous application of critics enables Argo to provide timely feedback. Criticism control mechanisms help make continuous critiquing practical and reduce distractions (i.e., unneeded context switches) due to irrelevant feedback. In addition to improving design decisions, timely feedback helps the architect make timely process decisions, e.g., "is this design excursion complete?" and "does a past decision need reconsideration?"

5.3.   Comprehension and Problem Solving

5.3.1.   Theory

The theory of *comprehension and problem solving* observes that designers must bridge a gap between their mental model of the problem or situation and the formal model of a solution or system (Kintsch and Greeno, 1985; Fischer, 1987). The *situation model* consists of designers' background knowledge and problem-solving strategies related to the current problem or design situation. The *system model* consists of designers' knowledge of an appropriate formal description. Problem solving or design proceeds through successive refinements of the mapping between elements in the design situation and elements in the formal description. Successive refinements are equated with increased comprehension, hence the name of the theory.

In the domain of software, designers must map a problem design situation onto a formal specification or programming language (Pennington, 1987; Soloway and Ehrlich, 1984). In this domain, the situation model consists of knowledge of the application domain and programming plans or design strategies for mapping appropriate elements of the domain into a formal description. The system model consists of knowledge of the specification or programming language's syntax and semantics. Programming plans or design strategies enable designers to successively decompose the design situation, identify essential elements and relationships, and compose these elements and relationships into elements of a solution. At successive steps, designers can acquire new information about the situation model or about the system model.

Pennington observed that programmers benefited from multiple representations of their problem and iterative solutions (Pennington, 1987). Namely multiple representations such as program module decomposition, state, and data flow, enabled programmers to better identify elements and relationships in the problem and solution and, thus, more readily to create a mapping between their situation models and working system models. Kintsch and Greeno's research indicated that familiar aspects of a situation model improved designers' abilities to formulate solutions (Kintsch and Greeno, 1985). These two results were applied and extended in Redmiles' research on programmers' behavior, where again multiple representations supported programmers' comprehension and problem solving when working from examples (Redmiles, 1993).

Dividing the complexity of the design into multiple perspectives allows each perspective to be simpler than the overall design. Moreover, separating concerns into perspectives allows information relevant to certain related issues to be presented together in an appropriate notation (Robbins et al., 1996). Design perspectives may overlap: individual design materials may appear in multiple perspectives. Coordination among design perspectives ensures that materials and relationships presented in multiple perspectives may be consistently viewed and manipulated in any of those perspectives. Overlapping, coordinated perspectives aid understanding of

new perspectives because new design materials are shown in relationship to familiar ones (Redmiles, 1993).

Good designs usually have organizing structures that allow designers to locate design details. However, in complex designs the expectation of a single unifying structure is a naive one. In fact, 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. Alternative decompositions of the same complex design can support the organizing structures that arise from these forces and the different mental models of stakeholders with differing backgrounds and interests. Using diverse organizing structures supports communication among stakeholders with diverse backgrounds and mental models which is key to developing complex systems that are robust and useful.

It is our contention that no fixed set of perspectives is appropriate for every possible design; instead perspective views should emphasize what is currently important in the project. When new issues arise in the design, it may be appropriate to use a new perspective on the design to address them. While we emphasize the evolutionary character of design perspectives, an initial set of useful, domain-oriented perspectives can often be identified ahead of time (Fischer et al., 1994).

5.3.2. *Support in Argo*

Multiple, overlapping design perspectives in Argo allow for improved comprehension and problem solving through the decomposition of complexity, the leveraging of the familiar to comprehend the unfamiliar, and the use of notations appropriate to multiple stakeholders' interests. Supporting the mental models of a particular domain must be done by domain engineers, practicing architects, and other stakeholders who apply Argo to a specific domain. Architects and other stakeholders may define their own perspectives in the course of design. Presentation and evolvability critics advise architects in defining and using perspectives.

Soni, Nord, and Hofmeister 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; and (4) code architecture describes the way that source code and other artifacts are organized in the development environment (Soni, Nord, and Hofmeister, 1995). Their experience indicates that separating out the concerns of each view leads to an overall architecture that is more understandable and reusable.

The 4+1 View Model (Kruchten, 1995) consists of four main views: (1) the logical view describes key abstractions (classes) and their relationships, e.g., *is_a* and *instantiates*; (2) the process view describes software components, how they are grouped into operating system processes, and how those processes communicate; (3) the development view describes source code modules and their dependencies; (4) the physical view describes how the software will be distributed among processors during execution. These four views are supplemented with scenarios and use

cases that describe essential requirements and help relate elements of the various views to each other. The views provide a well-defined model of the system, but more importantly they identify and separate major concerns in software development. The Unified Modeling Language (UML) also uses multiple perspectives to visualize various aspects of a design (Fowler and Scott, 1997). In demonstrating Argo, we chose perspectives similar to those described in these approaches; however, we believe that the choice of perspectives depends on the type of software being built and the tasks and concerns of design stakeholders.

Argo supports multiple, coordinated perspectives with customization. In addition to the perspectives described in this paper, Argo allows for the construction of new perspectives and their integration with existing perspectives. Architects who are given a fixed set of formal notations often revert to informal drawings when those notations are not applicable (Soni, Nord, and Hofmeister, 1995). 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, we allow architects to annotate perspectives with arbitrary, unstructured graphics (as demonstrated in Figure 3). Customizable presentation graphics are needed because the unifying structures of the system under construction must be communicated convincingly to other architects and system implementors. To be convincing, the style of presentation must fit the professional norms of the development organization: it should look like a presentation, not an architect's scratch pad. Furthermore, ad-hoc annotations that are found to be useful can be incrementally formalized and incorporated into the notations of future designs (Shipman and McCall, 1994). We expect that Argo's low barrier to customization will encourage evolution from unstructured notations to structured ones as recurring formalization needs are identified.

6.   Evaluation

The preceding section has provided theoretical evaluation of our extensions to the DODE approach. Also, the implementation of Argo described in Section 4 provides a proof-of-concept that many of the desired features for Argo can be realized. This section outlines our plans to further evaluate Argo as a working tool. Argo's architecture and infrastructure can be evaluated in terms of how well they support construction of design environments in new domains. Argo's support for cognitive needs can be evaluated by measuring qualities of design processes and products.

6.1.   Application to New Domains

The process of applying Argo to a new domain consists of defining new design materials with critics, a design process, and design perspectives. Below we describe our experience in carrying out these tasks for three domains.

In the domain of C2-style software architectures, there are two basic design materials: software components and connectors. The basic relationship between these materials describes how they communicate. This basic model was extended to include design materials for operating system threads, operating system processes, and source code modules. We rapidly authored approximately twenty critics that check for completeness and consistency of the representation and adherence to published C2 style guidelines (Taylor et al., 1996). The number of needed critics is small because the C2 style addresses only system topology and simple communication patterns. The C2 design process started with tasks to create each of the design material types, and was refined by splitting activities based on possible design material attributes, e.g., reused components vs. new components. We started with two perspectives discussed in previous work on C2, conceptual and implementation, and later included a perspective to visualize relationships between software components and the program modules that implement them.

We have also adapted the Argo infrastructure to implement a design environment for object-oriented design that supports a subset of the Object Modeling Technique (Rumbaugh et al., 1991). Since this domain is well defined and described in a single book, it was straightforward to identify the design materials, relationships, graphical notations, and perspectives. Our OMT subset includes the object model, behavioral model, and information model, but excludes the more advanced features of each. A core set of ten critics that address correctness and completeness of the design was also straightforward to implement, e.g., an abstract class with no subclasses indicates an incomplete design. Additional critics were inspired by a book on OO design heuristics (Riel, 1996), e.g., a base class should not make direct references to its subclasses because that means that adding new subclasses requires modifications to the base class. Some of these heuristics were more difficult to specify as critics because they rely on information not present in the representation, e.g., semantically related data and behavior should be kept together. In this example, an authoritative answer cannot be given because the OMT design representation does not contain enough semantic information; however, critics may apply pessimistic heuristics to identify when this issue might be a problem. The provided process and perspectives collected various process fragments described in these two books.

We have begun to apply Argo to software requirements specifications using the CoRE methodology (Faulk et al., 1992) in the avionics application domain. CoRE is based on the SCR requirements methodology (Henninger, 1980) with extensions that deal with the modular decomposition of the requirements document. As with OMT, existing documents describe the design materials, standard notations, and analyses. Existing tools cover essentially all analyses that can be performed on the requirement specification without considering the application domain, e.g., identifying non-deterministic transitions in a mode-transition table. In implementing a CoRE design environment we will demonstrate added value over existing tools by integrating analysis more tightly with the cognitive process of devising a specification, and by providing heuristics to support modularization of requirements

documents in the avionics domain, e.g., autopilot control modes are largely independent of cockpit display modes and should be specified in separate requirements modules, however there should be certain constraints between the two. To date we have implemented twenty critics that check correctness and completeness of CoRE requirements specifications and integrated them into an independently developed requirements editing tool. Doing so has given us additional confidence in our critiquing infrastructure.

Argo's architecture and infrastructure have provided satisfactory support for the initial implementations of domain-oriented design environments in three domains. We plan to evaluate how well our infrastructure extends in three dimensions: (1) larger domains with more critics and more complex designs and processes, (2) addition of new domain-oriented plug-ins (e.g., design rationale), and (3) use of the infrastructure by people outside of our research group (e.g., an avionics software development group).

6.2. Evaluating Cognitive Support

To evaluate Argo's support for the cognitive needs of designers, user testing will focus on how Argo affects the productivity of the designer and the quality of the resulting product. Our support for reflection-in-action should increase productivity by decreasing time spent reworking design decisions, lead to better designs in cases where critics provide knowledge that the designer lacks, shorten the lifespan of errors, reduce the number of missing design attributes, and strengthen the designer's confidence in the final design because more issues will have been raised and addressed. We expect our support for opportunistic design will allow designers to rely less on mental or paper notes, and to make better process choices. Comprehension of a sample design should increase when the designer's mental models match one or more design perspectives. Some experimental data can be automatically collected, e.g., the lifespan of errors, while others will rely on human observation and interviews. Experimental subjects will use Argo with all features enabled, while control subjects will use Argo with some features disabled. We plan to evaluate the resulting designs with the help of blind judges, as described by Murray (Murray, 1991). Tests that measure our hypotheses will indicate the degree to which identified cognitive needs are supported by Argo's features, and thereby suggest weights for the associations in Table 3.

A related task is devising a methodology for on-going evaluation of the quality of the knowledge provided by critics, the guidance contained in process models, and the mental models suggested by perspectives. This methodology should support on-going maintenance of the design environment and periodic reorganization and "reseeding" of domain knowledge (Fischer et al., 1994). Structured email between designers and knowledge providers is one source of data for this evaluation. We are also investigating event monitoring techniques that capture data to help evaluate how well provided knowledge impacts actual usage. Examples of quantities that could be automatically collected include the number of critics that fire, how many

"to do" items the designer views, and how many errors are fixed as a result of viewing feedback from critics.

A recent evaluation of VDDE (Voice Dialog Design Environment) raised several questions about the character of the impact of design critics (Sumner, Bonnardel, and Kallak, 1997). The study found that designers preempted critical feedback by anticipating criticisms and avoiding errors that the critics could identify. Designers assessed the relevance of each criticism before taking action, and in cases where experienced designers disagreed with criticism they usually added design rationale describing their decision. Sumner, Bonnardel, and Kallak suggest that evaluation of critiquing systems should explicitly consider designers of differing skill levels. They further suggest that future critiquing systems use alternative interface metaphors that users will perceive as cooperative rather than adversarial. In our own experiments we plan to group subjects by experience and watch closely for anticipation of criticism.

7.   Conclusions and Future Work

Designing a complex system is a cognitively challenging task; thus, designers need cognitive support to create good designs. In this paper we have presented the architecture and facilities of Argo, our software architecture design environment. Argo's architecture is motivated by the desire for reuse and extensibility. Argo's facilities are motivated by the observed cognitive needs of designers. The architecture separates domain-neutral code from domain-oriented code and active design materials. The facilities extend previous work in design environments by enhancing support for reflection-in-action, and adding new support for opportunistic design, and comprehension and problem solving.

In future work, we will continue exploring the relationship between cognitive theories and tool support. Further identification of the cognitive needs of designers will lead to new design environment facilities to support those needs. Also, we will seek ways to better support the needs that we have identified in this paper, e.g., a process model that approximates the cognitive cost of switching design tasks. Furthermore, we will investigate ways of better supporting and using design rationale. For example, the architect's interactions with the "to do" list is a potentially rich source of data for design rationale: items are placed on the list to identify open issues, and removed from the list as those issues are resolved. Design rationale is an important part of design context and "to do" items should reference relevant past items when possible.

Our current prototype of Argo is robust enough for experimental usage. It is our goal to develop and distribute a reusable design environment infrastructure that others may apply to new application domains. Successful use of our infrastructure by others will serve to inform and evaluate our approach. A Java version of Argo with documentation, source code, and examples is available from the authors.

Acknowledgments

Notes

1. KLAX is a trademark of Atari Games.

References

Ackerman, M. S. and McDonald, D. W. 1996. Answer Garden 2: Merging Organizational Memory with Collaborative Help. *Proc. ACM Conf. on Computer Supported Cooperative Work (CSCW'1996)*. pp. 97–105.

Abowd, G., Allen, R., and Garlan, D. 1993. Using Style to Understand Descriptions of Software Architecture. *SIGSOFT Software Eng. Notes*. vol. 18, no. 5. pp. 9–20.

Allen, G. and Garlan, D. 1994. Beyond Definition/Use: Architectural Interconnection. *Workshop on Interface Definition Languages*, published in *SIGPLAN Notices*. vol. 29, no. 8. pp. 35–45.

Batory, D. and O'Malley, S. 1992. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. Software Eng. and Methodology*. vol. 1, no. 4. pp. 355–398.

Benner, K. M. 1996. Addressing Complexity, Coordination, and Automation in Software Development with the KBSA/ADM. *Proc. 11th Knowledge-Based Software Eng. Conf.* Syracuse, NY. pp. 73–83.

Cugola, G., Di Nitto, E., Ghezzi, C., and Mantione, M. 1995. How to Deal with Deviations During Process Model Enactment. *Proc. 1995 Int. Conf. on Software Eng. (ICSE'95)*. Seattle, WA, April 23-30, 1995. pp. 265–273.

Curtis, W., Krasner, H., and Iscoe, N. 1988. A Field Study of the Software Design Process for Large Systems. *Comm. ACM*. vol. 31, no. 11. pp. 1268–1287.

Dourish, P., Holmes, J., MacLean, A., Marqvardsen, P., Zbyslaw, A. 1996. Freeflow: Mediating Between Representations and Action in Workflow Systems. *Proc. ACM Conf. on Computer Supported Cooperative Work (CSCW'96)*. Cambridge, MA. pp. 190–198.

Engelbart, D. 1988. 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. pp. 35–66.

Engelbart, D. 1995. Toward Augmenting the Human Intellect and Boosting our Collective I.Q. *Comm. ACM*. vol. 38, no. 8. pp. 30–33.

Faulk, S., Bracket, J., Ward, P., Kirby, Jr., J. 1992. The CoRE Method for Real-Time Requirements. *IEEE Software*. vol. 9, no. 9. pp. 22–33.

Fischer, G. 1987. Cognitive View of Reuse and Redesign. *IEEE Software, Special Issue on Reusability*. vol. 4, no. 4. pp. 60–72.

Fischer, G. 1994. Domain-Oriented Design Environments. *Journal of Automated Software Eng.* vol. 1, no. 2. pp. 177–203.

Fischer, G., Girgensohn, A., Nakakoji, K., and Redmiles, D. 1992. Supporting Software Designers with Integrated Domain-Oriented Design Environments. *IEEE Trans. on Software Eng.* vol. 18, no. 6. pp. 511–522.

Fischer, G. and Lemke, A. C. 1988. Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication. *Human-Computer Interaction*. vol. 3, no. 3. pp. 179–222.

Fischer, G., McCall, R., Ostwald, J., Reeves, B., and Shipman, F. 1994. Seeding, evolutionary growth and reseeding: supporting the incremental development of design environments. *Human Factors in Computing Systems, CHI '94 Conf. Proc.* Boston, MA. pp. 292–298.

Fischer, G., Nakakoji, K., Ostwald, J., Stahl, G., and Sumner, T. 1993. Embedding Computer-Based Critics in the Contexts of Design. *Human Factors in Computing Systems, INTERCHI '93, Conf. Proc.* Amsterdam, The Netherlands. pp. 157–164.

Fowler, M. and Scott, K. 1997. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley: Reading, MA.

Gantt, M. and Nardi, B. A. 1992. Gardeners and gurus: patterns of cooperation among CAD users. *Human Factors in Computing Systems, CHI'92 Conf. Proc.* pp. 107–117.

Garlan, D., Ed. 1995. *Proc. First Int. Workshop on Architecture for Software Systems*. Seattle, WA.

Garlan, D., Allen, R., and Ockerbloom, J. 1994. Exploiting Style in Architectural Design Environments. *Proc. Second ACM SIGSOFT Symposium on the Foundations of Software Eng.* Los Angles, CA. vol. 19, no. 5. pp. 175–188.

Garlan, D. and Shaw, M. 1993. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing.

Glance, N. S., Pagani, D. S., and Pereschi, R. 1996. Generalized Process Structure Grammars (GPSG) for Flexible Representations of Work. *Proc. ACM Conf. on Computer Supported Cooperative Work (CSCW'96)*. Cambridge, MA. pp. 180–189.

Green, C., Luchham, D., Balzer, R., Cheatham, T., and Rich, C. 1983. Report on a Knowledge-Based Software Assistant. RADC TR 83–195, Rome Laboratory.

Guindon, R. 1992. Requirements and Design of DesignVision, an Object-Oriented Graphical Interface to an Intelligent Software Design Assistant. *Human Factors in Computing Systems, CHI '92 Conf. Proc.* pp. 499–506.

Guindon, R., Krasner, H., and Curtis, W. 1987. Breakdown and Processes During Early Activities of Software Design by Professionals. In: Olson, G. M. and Sheppard S., eds. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing Corporation. pp. 65–82.

Hayes-Roth, B. and Hayes-Roth, F. 1979. A Cognitive Model of Planning. *Cognitive Science*. vol. 3, no. 4. pp. 275–310.

Henninger, K. L. 1980. Specifying Software Requirements for Complex Systems: New Techniques and Their Application. *IEEE Trans. Software Eng.* vol. 6, no. 1. pp. 2–14.

Int. Federation for Information Processing (IFIP). 1993. *Integration Definition for Function Modeling (IDEF0)*. Draft Federal Information Processing Standards Publication 183.

Kintsch, W. and Greeno, J. G. 1985. Understanding and Solving Word Arithmetic Problems. *Psychological Rev.* vol. 92. pp. 109–129.

Kruchten, P. 1995. The 4+1 View Model of Architecture. *IEEE Software*. vol. 12, no. 11. pp. 42–50.

Lemke, A. C. and Fischer, G. 1990. A Cooperative Problem Solving System for User Interface Design. *Proc. AAAI-90, Eighth National Conf. on Artificial Intelligence*. Cambridge, MA. pp. 479–484.

Lowry, M., Philpot, A., Pressburger, T., and Underwood, I. 1994. A Formal Approach to Domain-Oriented Software Design Environments. *Proc. 9th Knowledge-Based Software Eng. Conf.* Monterey, CA. pp. 48–57.

Luckham, D. C. and Vera, J. 1995. An Event-Based Architecture Definition Language. *IEEE Trans. Software Eng.* vol. 21, no. 9. pp. 717–734.

Medvidovic, N., Oreizy, P., and Taylor, R. N. 1997. Reuse of Off-the-Shelf Components in C2-Style Architectures. *Proc. 1997 Int. Conf. on Software Eng. (ICSE'97)*. Boston, MA, May 17-23, 1997. pp. 692–700.

Moriconi, M., Qian, X., and Riemenschneider, R. A. 1995. Correct Architecture Refinement. *IEEE Trans. Software Eng.* vol. 21, no. 4, pp. 356–372.

Murray, K. S. 1991. KI: A Tool for Knowledge Integration. *Proc. Thirteenth National Conference on Artificial Intelligence (AAAI'91)*. Portland, OR. vol. 1. pp. 835–842.

Partsch, H. and Steinbruggen, R. 1983. Program transformation systems. *ACM Computing Surveys.* vol. 15, no. 3. pp. 199–236.

Pennington, N. 1987. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology.* vol. 19. pp. 295–341.

Perry, D. E. and Wolf, A. L. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes.* vol. 17, no. 4. pp. 40–52.

Redmiles, D. F. 1993. Reducing the Variability of Programmers' Performance Through Explained Examples. *Human Factors in Computing Systems, INTERCHI '93 Conf. Proc.* Amsterdam, The Netherlands. pp. 67–73.

Rettig, M. 1993. Cooperative Software. *Comm. ACM.* vol. 36, no. 4. pp. 23–28.

Riel, A. 1996. *Object-Oriented Design Heuristics.* Addison-Wesley: Reading, MA.

Robbins, J. E., Morley, D. J., Redmiles, D. F., Filatov, V., and Kononov, D. 1996. Visual Language Features Supporting Human-Human and Human-Computer Communication. *Proc. 1996 IEEE Symposium on Visual Languages.* pp. 247–254.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. *Object-Oriented Modeling and Design.* Englewood Cliffs, NJ: Prentice Hall.

Schoen, D. 1983. *The Reflective Practitioner: How Professionals Think in Action.* New York: Basic Books.

Schoen, D. 1992. Designing as Reflective Conversation with the Materials of a Design Situation. *Knowledge-Based Systems.* vol. 5, no. 1. pp. 3–14.

Shipman, F. and McCall, R. 1994. Supporting Knowledge-Base Evolution with Incremental Formalization. *Human Factors in Computing Systems, CHI '94 Conf. Proc.* Boston, MA. pp. 285–291.

Silverman, B. and Mezher, T. 1992. Expert critics in engineering design: lessons learned and research needs. *AI Magazine.* Spring 1992. pp. 45–62.

Soloway, E. and Ehrlich, K. 1984. Empirical Studies of Programming Knowledge. *IEEE Trans. Software Eng.* vol. 10, no. 5. pp. 595–609.

Soloway, E., Pinto, J., Letovsky, S., Littman, D., and Lampert, R. 1988. Designing Documentation to Compensate for Delocalized Plans. *Comm. ACM.* vol. 31, no. 11. pp. 1259–1267.

Soni, D., Nord, R., and Hofmeister C. 1995. Software Architecture in Industrial Applications. *Int. Conf. Software Eng. 17.* Seattle, WA. pp. 196–207.

Sumner, T. 1997. The Cognitive Ergonomics of Knowledge-Based Design Support Systems. *Human Factors in Computing Systems, CHI '97 Conf. Proc.* pp. 83–90.

Taylor, R. N., Medvidovic, N., Anderson, K., Whitehead, Jr., E. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. 1996. A Component and Message-based Architectural Style for GUI Software. *IEEE Trans. Software Eng.* vol. 22, no. 6. pp. 390–406.

Terveen, L. G., Selfridge, P. G., and Long, M. D. 1993. From "Folklore" to "Living Design Memory." *Human Factors in Computing Systems, INTERCHI '93 Conf. Proc.* Amsterdam, The Netherlands. pp. 15–22.

Terveen, L., Stolze, M., and Hill, W. 1995. From "Model World" to "Magic World": Making Graphical Objects the Medium for Intelligent Design Assistance. *SIGCHI Bulletin.* vol. 27. no. 4. pp. 31–43.

Thomas, I. and Nejmeh, B. 1992. Definitions of Tool Integration for Environments. *IEEE Software.* vol. 9, no. 2. pp. 29–35.

Visser, W. 1990. More or Less Following a Plan During Design: Opportunistic Deviations in
   Specification. *Int. J. Man-Machine Studies.* pp. 247–278.