Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML

Jason E. Robbins and David F. Redmiles

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
Tel: 1 (949) 824-7308 Fax: 1 (949) 824-1715
{jrobbins,redmiles}@ics.uci.edu

Abstract
Software design is a cognitively challenging task. Most software design tools provide support for editing, viewing, storing, and transforming designs, but lack support for the essential and difficult cognitive tasks facing designers. These cognitive tasks include decision-making, decision ordering, and task-specific design understanding.

This paper describes Argo/UML, an object-oriented design tool using the Unified Modeling Language design notation. Argo/UML supports several identified cognitive needs of software designers. This support is provided in the form of design tool features. We describe each feature in the context of Argo/UML and provide enough detail to enable other tool builders to provide similar support in their own tools. We also discuss our implementation of the UML and XMI standards, and our development approach.

Keywords: UML, XMI, Cognitive support, Open-source software.

1. Introduction

Software designers have used diagrammatic representations of their designs since the earliest days of software development. Over time the nature of these design diagrams has changed and so have the tools used to produce them. Much like early word processors replaced typewriters, early CASE (Computer Aided Software Engineering) tools served as electronic replacements for paper, pencil, and stencil. Many of these early CASE tools became "shelfware" because they did not provide significant value to software designers. Later CASE tools added sophisticated code generation, reverse engineering, and version control features. These features add value via increased automation of some design tasks, for example, converting a design into a source code skeleton. However, current CASE tools fail to address the essential cognitive challenges facing software designers.

Software design is not simply an automatable process of transforming one specification into another; it also involves complex decision making tasks that require the attention of skilled designers. Design tools that support designers in decision-making are a promising way to increase designer productivity and the quality of the resulting designs.

Helping designers make good design decisions is important because these design decisions will strongly influence the amount of implementation and maintenance effort needed later. Support for designers is also important because many software designers are overworked and pressured to attempt design tasks for which they lack proper training and experience. This is due, in part, to the current shortage of trained information technology workers.

In this paper we present a set of novel design tool features intended to support design tasks. Each of these features is motivated by our experience in designing software systems and by published theories of the cognitive challenges of design. Using cognitive theories to guide the development of design tool features has resulted in several promising new features that we would not otherwise have been likely to invent. Section 2 briefly covers the theories and previous work that lead to these features. Section 3 presents several specific cognitive features.

Each feature is described in the context of Argo/UML, a tool for object-oriented design that uses the Unified Modeling Language. Argo/UML is a research system with an emphasis on novel features, however, it includes enough standard CASE tool functionality to be generally useful. It supports UML class diagrams, state diagrams, use case diagrams, activity diagrams, and collaboration diagrams. It can generate Java™ source code from class diagrams. Argo/UML is implemented in Java and consists of about 100,000 lines of code in 800 classes. Preliminary versions of Argo/UML have been

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and F30602-94-C-0218, and by the National Science Foundation under Contract Number CCR-9624846. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory or the U.S. Government.

distributed via our web site (www.ics.uci.edu/pub/arch/uml) since July 1998, and have been evaluated or used in dozens of companies and universities. Argo/UML is the fourth in a series of tools that we have developed or enhanced with cognitive support features. Although Argo/UML is the focus of this paper, the cognitive support features we describe are also applicable to other design tools. We will return to this point in the conclusion of this paper. Section 4 presents the implementation of Argo/UML.

The development of Argo/UML is the result of approximately two person-years of effort. This level of productivity was achieved, in part, by taking advantage of appropriate standards, namely, the Unified Modeling Language (UML), the XML Model Interchange format (XMI), and the Java component model (JavaBeans). Also, development of Argo/UML has followed the open-source strategy of encouraging early users to become co-developers. Section 5 discusses our development approach in more detail.

2. Background and related work

UML [22] is a standard object-oriented design notation that combines work done previously on the Booch notation, OMT (Object Modeling Technique), and OOSE (Object-Oriented Software Engineering). Standardization produces an economy of scale that leads to more and better tools, better interoperability between tools, more developers who are skilled in using the standard notation, and lower overall training costs. XMI is a standard file format for UML designs [25].

Design critics are active agents that reside in a design tool and continuously check the design for potential errors, stylistic violations, and incomplete sections. Design critics have been used in design tools for many domains, including building architecture [4, 5, 6], user interfaces [3, 13, 14], and medical diagnostics [8, 12]. Design critics provide knowledge support to designers who lack specific pieces of knowledge about the problem or solution domains, e.g., beginning designers or expert designers who are expanding their design vocabulary. Design critics also help catch slips and mistakes that occur when designers are distracted or working under pressure. Much of the work on critics has been motivated by Schoen's theory of *reflection-in-action*, which basically says that designers intermix synthesis and analysis in their design process [19].

Classic waterfall development models assume that work progresses in a systematic fashion. In reality, software design is a creative process and designers often jump from idea to idea. Task switching by designers is not random, it can be explained by theories of *opportunistic design* [23]. For example, designers tend to switch tasks when they don't have the knowledge needed to continue on their current task or when knowledge for another task suddenly comes to mind.

Many CASE tools provide multiple views on the design; in fact, UML itself defines seven diagram types. We have tried to complement these standard design views with *task-specific views*. Research on problem understanding indicates that providing the right view can help designers bridge cognitive gaps between their mental models of the problem and solution [11, 16].

3. Overview of cognitive features

Figure 1 shows the Argo/UML user interface. Designers work with Argo/UML much as they would work with other CASE tools; in fact, we have tried to use standard user interface styles as much as possible to make Argo/UML more approachable. Design diagrams are edited in the large, upper-right pane. This large pane can also be used for table views of the design. Properties of the selected design element can be edited in the various property tabs located in the lower-right pane. The upper-left pane gives a tree-structured overview of the design, while the lower-left pane contains the designer's "to do" list.

The following subsections describe cognitive support features inspired by theories of cognition in design.

3.1. Critics and criticism control mechanisms

As mentioned above, design critics are agents that check the design for potential problems. In Argo/UML critiquing is done in a thread of control that is separate from the main application and user interface. Critiquing is done continuously and designers need not request that critics be applied or even know that any particular critic exists. Designers do not see critics; only the design feedback produced by critics is seen.

Critics can 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 problems or areas needing improvement in the design; only the most severe errors are prevented outright, thus allowing the designer to work through invalid intermediate states of the design.

Each critic performs its analysis independently of others, checking one predicate, and delivering one piece of design feedback. Some critics in Argo/UML are derived directly from constraints in the UML semantics specification [22]; for example, one critic checks that an association has at most one composite end. Others come from published guidelines on object-oriented design and design patterns [7, 17]. Corporate design guidebooks and best practices are a promising, organization-specific source of knowledge that could be actively delivered by critics. Lastly, some critics deal with limitations of the tool; for example, one critic warns that our Java code generator cannot handle multiple inheritance.

Formalizing the analyses and rules of thumb used by practicing software designers could produce hundreds of critics. Critics must be controlled so as to make efficient use of machine resources, but our primary focus is on effective interaction with the designer. Specifically, designers should be able to easily view relevant and timely feedback items without having to sort through

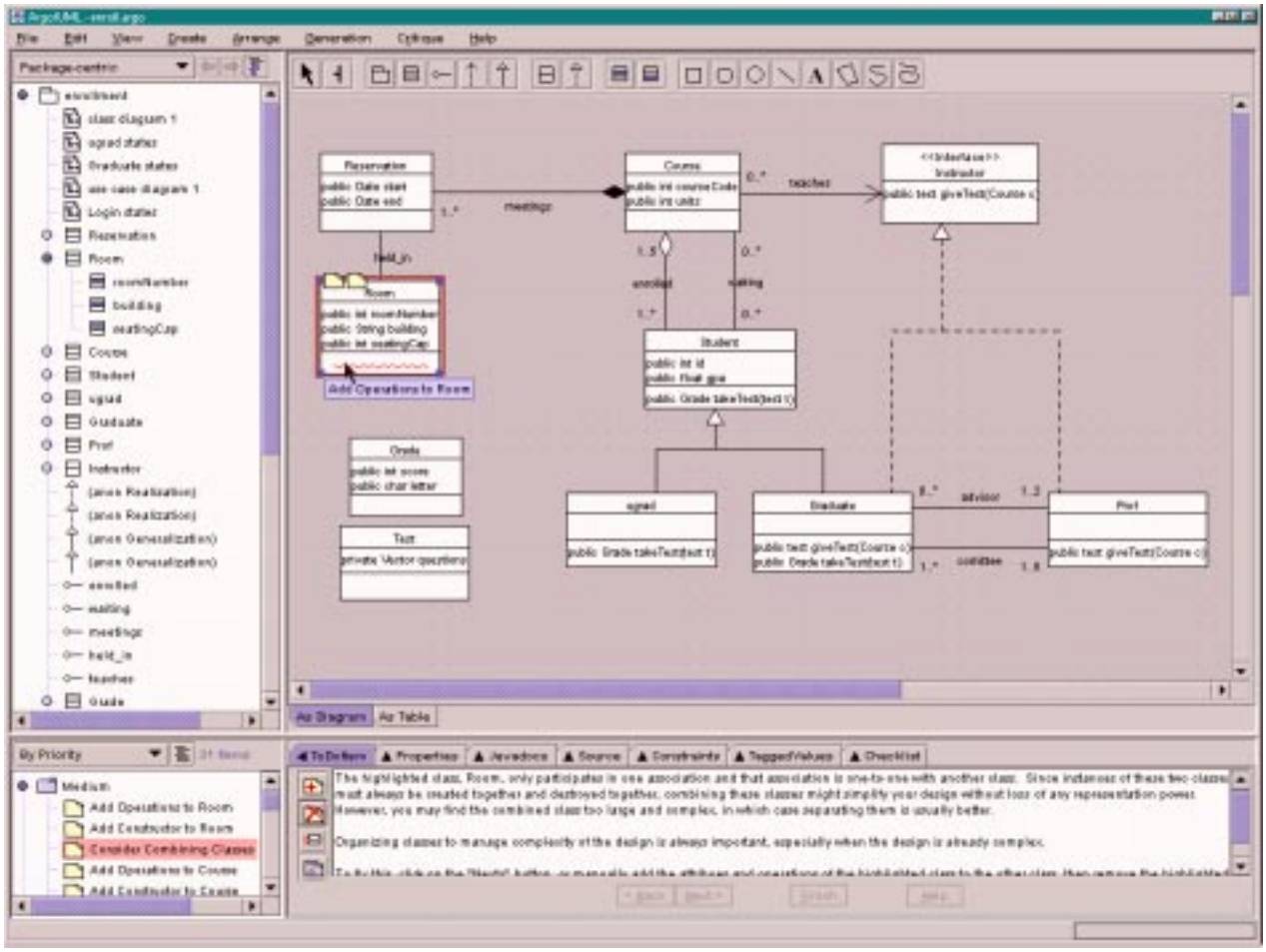


Figure 1. The Argo/UML user main window.

irrelevant items. Furthermore, the elapsed time between a design manipulation that introduces an error and the presentation of feedback identifying the error should be as short as possible, ideally within one second so as to maintain a feeling of interactivity.

Criticism control mechanisms are predicates used to limit execution of critics to when they are relevant and timely to decisions being considered by the designer. Attributes on each critic identify what type of goals and design decisions it supports. Criticism control mechanisms check those attributes against the user model to select critics for activation. Computing relevance and timeliness separately from critic predicates allows critics to focus entirely on identifying problematic conditions in the design product while leaving cognitive design process issues to the criticism control mechanisms.

3.2. "To do" list

Once critics generate design feedback, it must be presented to the designer in a usable form without causing distraction. Our "to do" list user interface presents feedback to the designer (Figure 2). The "to do" items on the list are grouped by category, for example, by priority, by design decision type, or by offending design element. When the designer selects a "to do" item from the lower-left pane, the associated (or "offending") design elements are highlighted in all diagrams and details about the identified problem and possible resolutions are displayed in the "ToDoItem" tab.

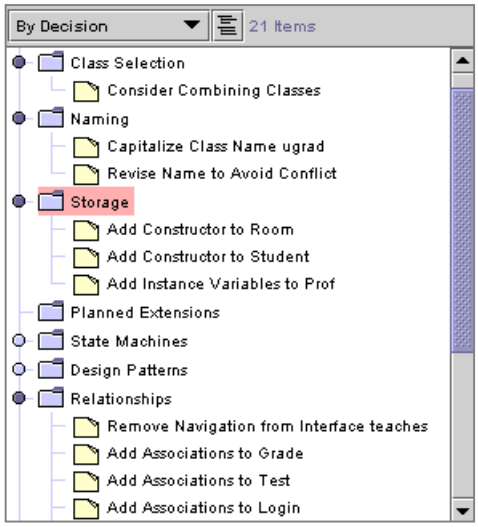


Figure 2. The designer's "to do" list.

Toolbar buttons in the "ToDoItem" tab allow the designer to add a new item as a personal reminder, follow links to background domain knowledge relevant to the issue at hand, snooze the critic (disable it for a limited time), send email to the person who authored the critic, or dismiss the item. Links to background information and email contact with critic authors provide a design context that helps the designer resolve the issue at hand. Providing contact information for relevant stakeholders helps to situate the problem and possible solutions in the context of the development organization.

User testing with an early version of Argo/UML demonstrated that designers are likely to focus on the diagram pane to the exclusion of the "to do" list pane. Designers were observed to build on incorrect design decisions despite the fact that criticism of those decisions was listed in another pane in the same window. We added *clarifiers* to Argo/UML to make criticism more evident to designers engaged in design construction. Clarifiers are icons or other visual indications of errors that are displayed directly on the design diagram [20]. For errors that occur at a specific part of a design element we use wavy, red underlines (a familiar indication of spelling errors). Errors that relate to an entire design element are shown with yellow sticky-note icons. We avoid clutter by only displaying clarifiers on the currently selected design element. Designers will encounter clarifiers in the normal course of manipulating the design with the mouse and keyboard. If the designer positions the mouse pointer over a clarifier, a feedback item headline is displayed as a tool-tip.

3.3. Non-modal wizards

Critics that simply identify problems leave the full responsibility for fixing those problems with the designer. Often when a critic identifies a specific problem, there is a specific, automatable solution to that problem. For example, one critic identifies class names that begin with lowercase letters as unconventional; a push-button solution to this problem is to capitalize the first letter of the class name. However, not all solutions can be done in a single step: some will require the designer to make decisions about how the problem should be resolved. For example, one Argo/UML critic identifies multiple inheritance as incompatible with Java code generation; the corrective steps involve choosing one superclass to convert into an interface and moving attributes down into subclasses.

Argo/UML uses non-modal wizards to aid designers in carrying out suggested design improvements. Argo/UML's wizards are similar to wizards found in other tools: they guide the designer through steps and decisions in a predefined task. A wizard typically performs design manipulations on the designer's behalf, however, some suggested fixes consist solely of step-by-step instructions to the designer. The designer uses "Next" and "Back" buttons to move among steps, and branches are taken based on the state of the design and the values entered into the wizard. As the designer progresses through the steps, a blue progress bar is drawn on the sticky-note icon for the affected feedback item in the "do to" list.

Unlike wizards found in other tools, our wizards are non-modal and apply changes immediately. The designer is free to leave a wizard at any time to directly manipulate the design or use another wizard. The designer may return to a partially completed wizard at any time. The ability to directly manipulate the design is needed for wizards that simply direct the designer through a series of steps that are not automated. The non-modal nature of our wizards also allows designers to opportunistically order decisions: if an idea occurs to them while working through a wizard, they are free to pursue the new idea without being locked into the wizard or fearing that they might forget to finish their current task.

3.4. Checklists

Design review meetings are one of the most cost-effective ways to remove defects. Many organizations have standardized their best practice by using design review checklists. Argo/UML provides electronic design checklists as a simple and flexible form of knowledge support (Figure 3). Argo/UML improves on paper checklists by presenting only those items relevant to the design element at hand and by phrasing general guidelines in concrete terms. For example, a paper checklist might have an item "Could this attribute be moved from this class to a superclass?" whereas an Argo/UML checklist item can be more concrete, e.g. "Could gradePointAvrg be moved from Undergraduate to

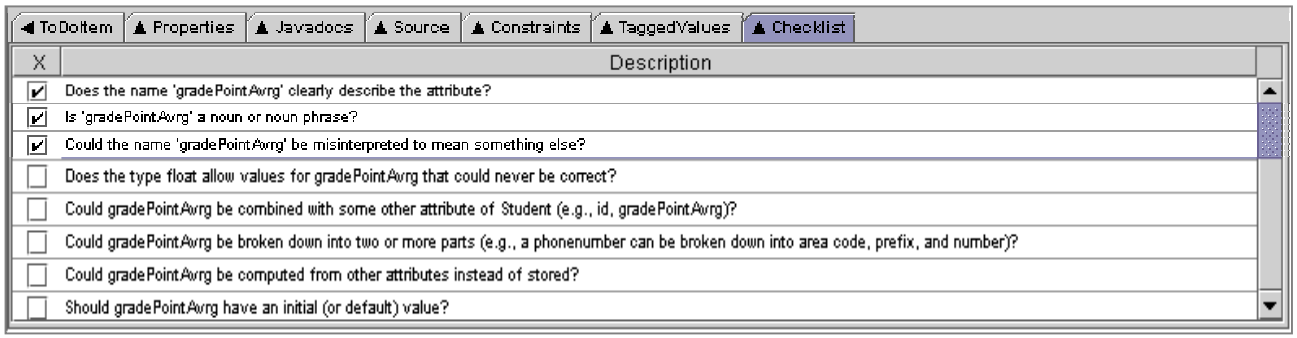


Figure 3. Checklist items for a UML attribute.

Table view of: Graduate states [Rows: 017]

Table: Transitions vs. Properties

Filter:

Name	Source	Target	Trigger	Guard	Effect
(anon Transition)	candidacy Exam	findTopic	committeeApproves		
(anon Transition)	findTopic	topic Defense	advisorApproves		
(anon Transition)	topic Defense	write Dissertation	committeeApproves		
(anon Transition)	write Dissertation	final Defense	getJob		
(anon Transition)	final Defense	final	pass		
(anon Transition)	quals	quals	fail		
(anon Transition)	take Classes	final	quit		
(anon Transition)	quals	final	quit		
(anon Transition)	survey	final	quit		
(anon Transition)	candidacy Exam	final	quit		
(anon Transition)	findTopic	final	quit		
(anon Transition)	topic Defense	final	quit		
(anon Transition)	write Dissertation	final	quit		

As Diagram | As Table

Figure 4. Table view of a state machine emphasizing transitions.

Student?"

Checklists, like critics, help designers identify design problems early and privately. Checklist items serve mainly to remind designers of common problems and issues, leaving the designer to do more of the determination of whether the potential problem is actually present. Critics, on the other hand, perform more of that determination automatically. Some checklist items that are found useful can be reimplemented as critics. We provide both critics and checklists to support a range of issues from informal to formal and to allow organizations to gradually invest in custom knowledge support.

3.5. Opportunistic table views

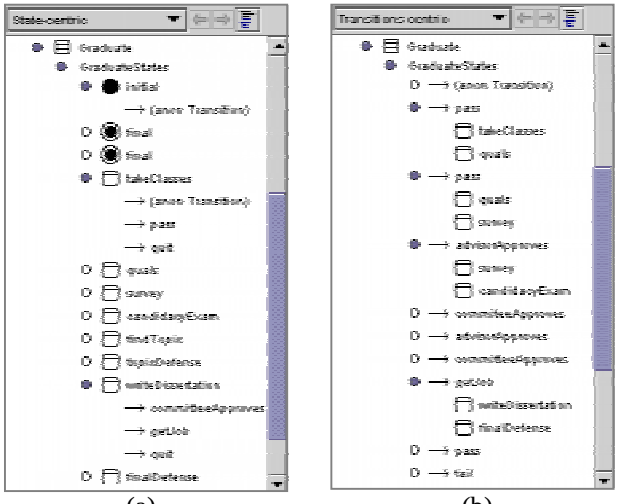
Argo/UML complements graphical design representations with task-specific table views. Each table view selects relevant attributes of design elements and presents them in a dense format. For example, one table view shows states as rows with the name, entry, and exit actions of each state in columns. Another table view shows the transitions as rows with the trigger, guard, effect, source, and destination as columns (Figure 4). As with navigational perspectives (described below), our goal is to provide views that support common design tasks. In particular, tables are much easier to systematically scan or fill-in than are diagrams.

However, designers work opportunistically as well as systematically. For example, if a designer is checking that each state has sensible entry and exit actions and finds one with a problematic assumption, he or she may switch to looking over the entire design for other elements which depend on that same assumption. These *design excursions* are natural and common; unfortunately, returning from an excursion imposes the cognitive difficulty of recalling one's prior plans. Argo/UML can help establish and recover systematic scanning behavior by highlighting blocks of table cells: as the designer moves the selected cell across rows or down columns, the entire row or column is highlighted. This highlighting fades gradually so that the designer can see what he or she was doing previously.

3.6. Navigational perspectives

Argo/UML, like most CASE tools, provides a navigation tree for the designer to access the various parts of the design. Unlike other tools, Argo/UML provides a much richer set of tree-structured perspectives and a customization language.

The designer can choose a *navigational perspective* from the menu above the navigation tree. For example, if the designer is working to define the possible states of a particular class, he or she might use the state-centric navigational perspective, which shows states as children of the state machine and transitions as children of states (Figure 5a). This emphasizes the states and makes the transitions secondary. Once the designer has a firm understanding of the states, he or she may wish to emphasize the transitions. The transition-centric



(a) (b)

Figure 5. Two navigational perspectives.

navigational perspective emphasizes transition by showing them as children of the machine and states as the children of transitions (Figure 5b).

We have defined several navigational perspectives that support various tasks in object-oriented software design. For each of these tasks we identify questions about the design that the designer must answer. One such perspective is the transition-paths perspective, it shows initial states as children of state machines and successor states as children of states. This helps the designer answer the question "if an object leaves this state, where can it go?" A related question is "how can an object get into this state?" We do not provide a predefined perspective to answer this question, but the designer can use a configuration window to define new perspectives to answer such questions as they arise.

Argo/UML's navigational perspective configuration window is shown in Figure 6. The top pane lists currently defined perspectives. The lower-left pane lists all predefined navigation rules, while the lower-right pane lists those navigation rules that are included in the selected perspective. Each navigation rule generates children of tree nodes. For example, the rule "State->Preceding States" will be applied to any tree node that represents a state and will generate a child for each state with a transition into the parent state. This rule could be used to answer the question of how an object can get into a given state. The set of possible navigation rules is large but finite; the UML standard meta-model includes about one hundred associations, each of which can have a corresponding navigation rule. Navigational perspectives are generated by applying all applicable rules whenever the user expands a tree node.

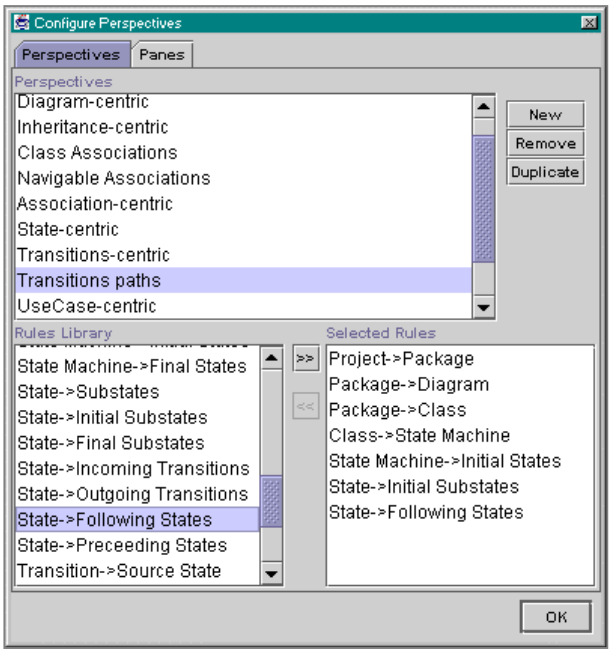
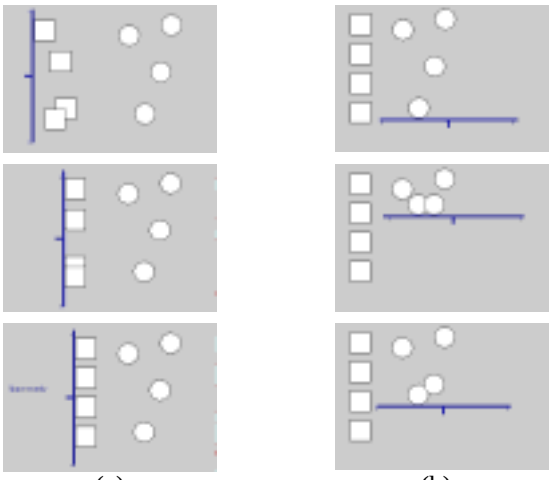


Figure 6. The perspective customization window.

3.7. Broom alignment tool

UML's syntax reserves the meaning of icons and text within a diagram. However, there are other visual aspects of diagrams that are left to the designer's discretion, including size, color, spacing, and alignment. Much as programmers use indentation and blank lines to express program structure, designers use the unreserved visual aspects of a diagram as a "secondary notation" that expresses relationships that are of concern but that are not covered by the formal notation [9]. Alignment is one important form of secondary notation.



(a) (b)

Figure 7. The broom alignment tool.

Our broom alignment tool pushes objects that come into contact with it, like a real-world push broom. This aligns objects along the face of the broom and provides immediate visual feedback. Pressing the space bar causes objects to be spaced evenly (Figure 7a). Unlike a real-world broom, moving backwards allows objects to return to their original position (Figure 7b).

We recently finished a user study of this user interface feature [18]. The study indicated that users moved and dragged the mouse substantially less when using the broom than when using standard alignment commands.

4. Architecture

The JavaBeans change notification design pattern [10] is the primary, recurring theme in Argo/UML's architecture (Figure 8). The top-level component is responsible for the rough layout of the user interface and the loading of features. The bottom-level component is our implementation of the standard UML meta-model. The middle layer consists of user interface features.

Integration between layers is generally loose. Each feature implements an API that defines its role in the user interface. The top-level component passes information to the features based on run-time type information that identifies their role, rather than having specific knowledge of individual features. Features work fairly independently, but share underlying models. These

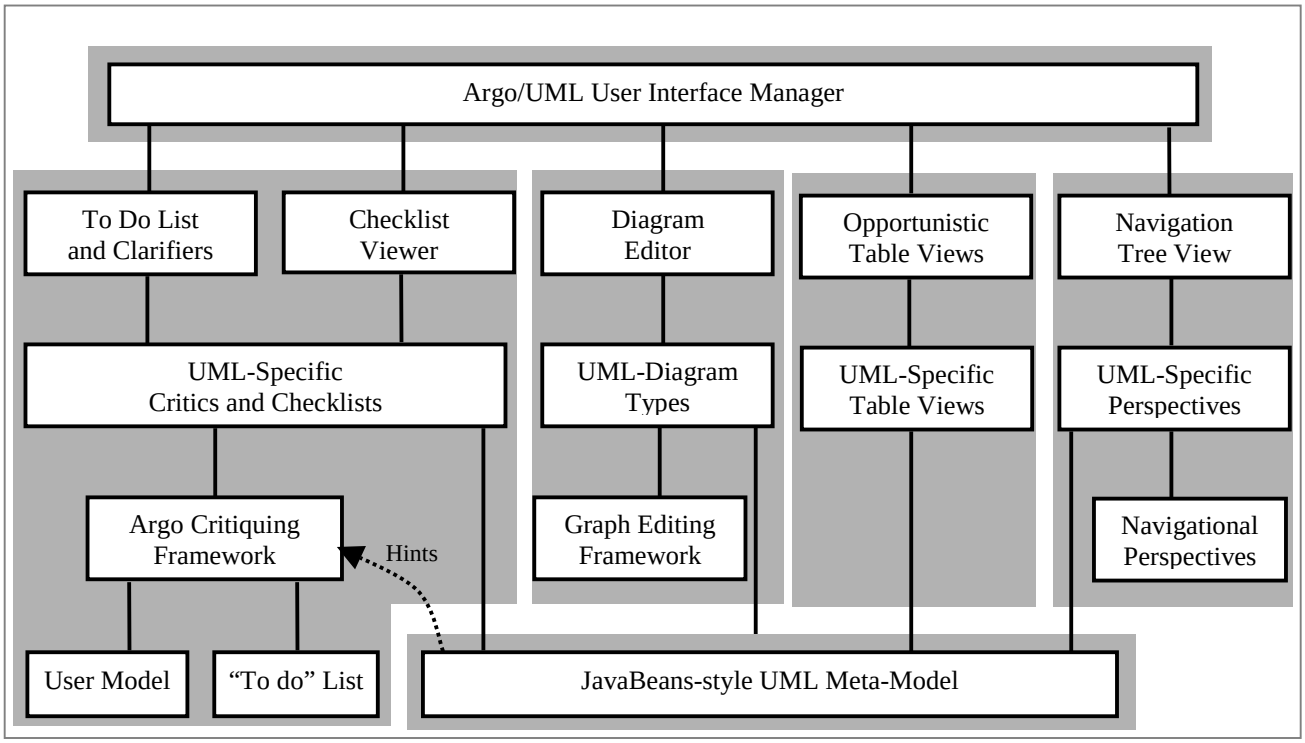


Figure 8. Argo/UML architecture.

shared models include the UML design representation, a user model, and the designer's "to do" list. Dependencies between the models and features are minimized and managed. For example, our critiquing and graph editing frameworks operate over networks of objects, without assuming that they are part of the UML meta-model. And in the other direction, our UML meta-model consists almost entirely of get-and-set methods that broadcast standard JavaBeans change notifications.

To enhance interactive performance, we have allowed some dependencies from the models to the features. However, each of these takes the form of an optional hint, and the features are implemented to work robustly in the absence of these hints. For example, most "set" methods in our UML meta-model also send a "critique as soon as possible" hint to the critiquing framework; however, all model elements will still be critiqued eventually, even without this hint.

4.1. UML meta-model and XMI

We generated our initial Java implementation of the UML meta-model by using a Rational Rose™ model provided with the UML 1.1 standard. As a result, our tool strictly adheres to the UML standard, including all the names of packages, meta-classes, attributes, and associations. Leveraging the standard helped to save development resources, which are very limited in an academic setting. Furthermore, strict adherence made it easier to support the XMI standard, which is itself based on the UML standard.

We made limited meta-model modifications to fit the Java language and our libraries. For example, we replaced multiple inheritance in the standard with Java interfaces and single inheritance. We also integrated JavaBeans-style method naming and change notifications into the meta-model. An assumption in GEF (described below) required us to add a Realization meta-class that is analogous to the Generalization meta-class. Recent proposed changes to the UML standard seems to better match our tool, and we expect to support UML 1.3 shortly after it is completed [21].

Argo/UML uses XMI files to store design representations. Using the XMI standard has helped us focus on cognitive issues by allowing us to defer issues of interoperability, version control, and repositories. We make use of IBM's XML parser to read XMI files using a straightforward set of tag handlers. We generate XMI files using a language we call TEE (Templates with Embedded Expressions). One TEE template is associated with each meta-class and consists of plain text to be echoed to the output file and embedded OCL (Object Constraint Language) expressions. Each OCL expression is applied to the design element and results in a bag of objects. Each result object is output in sequence and may use its own template. We also believe that TEE files could be an easily customizable way to generate Java source code. Table 1 gives some examples of the TEE files we use to generate XMI.

We chose not to use XMI's ViewElement, presentation, geometry, and style tags to represent diagrams. Instead we use the PGML (Precision Graphics Markup Language) standard file format for diagrams [1]. This has the advantage of being better defined, and may

allow users to view Argo/UML diagrams in future web browsers.

```
Template for Meta-class: Model
<Model XMI.id = '<ocl>self.id</ocl>'>
 <name><ocl>self.name</ocl></name>
 <visibility XMI.value
     ='<ocl>self.visibility</ocl>'/>
 <isAbstract XMI.value
     ='<ocl>self.isAbstract</ocl>'/>
 <isLeaf XMI.value='<ocl>self.isLeaf</ocl>'/>
 <isRoot XMI.value='<ocl>self.isRoot</ocl>'/>
 <ownedElement>
   <ocl>self.ownedElement</ocl>
 </ownedElement>
</Model>
```

```
Template for Meta-class: Class
<Class XMI.id = '<ocl>self.id</ocl>'>
 <name><ocl>self.name</ocl></name>
 <visibility XMI.value
     ='<ocl>self.visibility</ocl>'/>
 <isAbstract XMI.value
     ='<ocl>self.isAbstract</ocl>'/>
 <isLeaf XMI.value='<ocl>self.isLeaf</ocl>'/>
 <isRoot XMI.value='<ocl>self.isRoot</ocl>'/>
 <isActive XMI.value = 'false'/>
 <feature>
   <ocl>self.behavioralFeature</ocl>
   <ocl>self.structuralFeature</ocl>
 </feature>
 <taggedValue>
   <ocl>self.taggedValue</ocl>
 </taggedValue>
</Class>
```

Table 1. TEE templates for XMI.

4.2. Argo critiquing framework

The Argo critiquing framework is a major component of Argo/UML, it consists of about 5,000 lines of Java code in 40 classes. This framework provides default behavior for critics, schedules the application of critics, maintains a user model and the designer's "to do" list, and records each criticism and its resolution. One goal of this framework is to allow critic developers to start small and invest effort incrementally; for example, guidelines can be added easily as checklist items, refined into critics, and later supported by wizards with various degrees of automation. Another goal is to keep critiquing logic separate from the main application and robust in the case of incomplete information.

Critics are implemented as Java classes subclassed from class Critic. Class Critic defines several methods that may be overridden to define and customize a new critic. Each critic's constructor specifies the headline, problem description, and relevant decision categories. The central method is a predicate that accepts a design element to be critiqued and returns true if a problem is found. Most of the critics implemented in Argo/UML go no further than overriding this predicate. However, the default methods for generating a "to do" item and a clarifier can also be overridden. Another customizable aspect of critiquing is the determination of when an identified problem has been resolved. The headlines and descriptive text produced by critics contain embedded OCL expressions which are evaluated to specialize them to the offending design element; for example, the headline "Capitalize <ocl>self.name</ocl>" might become "Capitalize undergraduate", where "undergraduate" is the name of a class in the design. OCL expressions are also used to specialize checklist items.

Criticism control mechanisms are also implemented as Java classes that implement a predicate. This predicate accepts a critic and a user model and returns true if the critic should be enabled. Several criticism control mechanisms have been implemented and are jointly applied to the critics. All control mechanisms must agree that a critic should be enabled. For example, a relevance control mechanism checks that the critic supports a goal listed in the user model, and a timeliness control mechanism checks that the critic supports a design decision type listed in the user model.

The scheduling and application of critics is done in a critiquing thread of control, so as not to interrupt or slow down normal user interaction with Argo/UML. The intent of the scheduling algorithm is to minimize response time to design manipulations that introduce errors and to make productive use of otherwise idle computer time.

The critiquing thread executes an endless loop of three main steps: (1) recomputing the set of active critics, (2) applying critics to design elements in the "hot queue," (3) applying critics to a few design elements in the "warm queue." The overall CPU utilization of the critiquing thread is kept to an average of approximately twenty percent. The warm queue is essentially the open list of a standard breadth-first tree traversal that starts at the object representing the entire design project and eventually touches every design element. For most design projects this traversal would take longer than the desired interactive response time of one second, so it is done incrementally. The hot queue contains only design elements that are likely to generate new feedback and is typically short enough to be completely cleared in less than one second. Design elements are promoted to the hot queue in response to design manipulations that have the possibility of introducing problems. We further focus critiquing of hot queue elements by applying only those critics that have registered interest in the relevant type of design manipulation.

One key trade-off in a critic scheduling algorithm is the amount of knowledge the scheduler has about each critic. With no knowledge about what causes a particular critic to produce feedback, the scheduler can do no better than periodically applying all critics to all design elements. With complete knowledge about the analysis performed by individual critics, the scheduler can apply exactly those critics that will produce feedback as the result of a given design manipulation. Requiring less knowledge about critics helps to simplify the scheduler and reduce the development effort needed to author a

critic. Providing more knowledge about critics allows the critiquing system to work more efficiently and reduces feedback delays. In Argo/UML we require that all critics register interest in specific types of design elements and we allow critics to register interest in specific types of design manipulations. If the critic author does not define which design manipulations should trigger the critic, the critic will still be applied eventually via the warm queue. Likewise, if the design representation does not send hints to the critiquing scheduler when changes are made, the design elements will still be critiqued eventually.

We have found our critiquing framework flexible enough to build critiquing into four design and requirements tools. These include a state-based requirements tool, developed by a third-party, to which critiquing was added with no major modifications to its existing design representation. In Argo/UML we have implemented over sixty design critics.

4.3. Graph editing framework

GEF (Graph Editing Framework) is a major reusable component of Argo/UML. It consists of about 24,000 lines of Java code in 160 classes. Many software engineering tools include connected graphs in their user interfaces, and many researchers have developed connected graph editors. Two notable class frameworks for diagram and graph editing are HotDraw [2] and Unidraw [24]. We learned from these and decided to develop our own emphasizing extensibility, simplicity, and a high-quality user experience. HotDraw and Unidraw both achieve great extensibility by using flexible, abstract concepts. We limited the number and flexibility of our concepts to make our framework more understandable. Over time, we have applied GEF to many diagram types and enhanced the look and feel of GEF to provide a better user experience, but these extensions have not required us to generalize our basic concepts.

There are six major concepts in GEF. The Editor class acts as a hub that holds the other pieces together and routes messages among them. Figs (short for figures) are the primitive shapes; for example, FigCircle draws a circle and FigText draws text. Layers contain Figs in back-to-front order. Selections keep track of which Figs are selected and the effect of each handle; for example, SelectionResize allows the bounding box of a Fig to be resized, while SelectionReshape allows individual points of a FigLine or FigPoly to be moved. Cmds (short for commands) make modifications to the Figs; for example, CmdGroup removes the selected Figs from their Layer and adds a new FigGroup in their place. Modes are objects that process user input events (e.g., mouse movement and clicks) and execute Cmds to modify the Figs; for example, dragging in ModeSelect shows a selection rectangle, while dragging in ModeModify moves the selected objects. We have made central those concepts that are familiar to diagram editor users and avoided those that are unfamiliar or too abstract; for example, GEF does not use the decorator pattern [7] or attempt to offer general purpose constraint solving.

Initially, we implemented a generic connected graph representation as the underlying model for GEF diagrams. After using GEF in several applications we found existing data structures that could be interpreted as graphs; for example, Argo/UML has the UML meta-model. Now, the mapping from Figs in a diagram to application objects in an underlying data structure is managed by GraphModels. GraphModels themselves do not hold much data, rather they interpret existing data structures as graphs. For example, StateDiagramGraph-Model interprets UML States as nodes and UML Transitions as edges. Our GraphModels are analogous to mediators found in Java's Swing user interface library; for example, Swing tree widgets use TreeModels and table widgets use TableModels. As with other standards we have adopted, following Swing's design patterns has conserved our development resources.

Overall, we have found GEF's design to be flexible enough to define UML's diagram types, and extensible enough to accommodate novel interaction styles; for example, the broom alignment tool fitted in nicely as ModeBroom. GEF has been distributed via our web site since 1996 and used by dozens of universities and companies. Some developers using GEF have reported that it takes about three days to grasp the framework well enough to begin making extensions.

5. Development approach

It has been a great challenge to develop and support a product as large as Argo/UML in an academic setting. Since 1996 about two person-years of effort have gone into development, testing, documentation, and support. Students did essentially all of this work.

Three main factors allowed us to achieve so much. First, a very small team with the right tools can make huge strides, partly because of low communication overhead. Second, leveraging standards guides development and reduces the need to develop and document new approaches. Third, the open-source approach helps generate realistic requirements and motivate higher quality. Using standards and gathering feedback via the open-source approach both help avoid the risk that a small team will produce an idiosyncratic product.

Work on GEF and Argo/UML has followed the open-source approach from the start. As soon as an initial version of each library or product was completed, we built a web site to distribute the source code and documentation and invite feedback and code contributions. Ideally, early users of our code would have become co-developers, thus increasing development resources. To date, Argo/UML has about 2000 registered users in addition to classroom use. Unfortunately, we have received very few code contributions. This is partly due to the effort needed to encourage and coordinate remote developers; we now realize that managing an

open-source project is itself a full-time effort. We have, however, received a large volume of feedback on the way CASE tools are used, features needed in Argo/UML, and bug reports. Currently, we receive about five such email messages each day. This feedback has helped us directly in making a practical and desirable tool, and it has also helped indirectly by maintaining a product-oriented attitude in our academic environment.

6. Conclusion

In this paper we have described some of the cognitive features of Argo/UML and key parts of the system's design. The cognitive features provide designers with support for decision-making, decision ordering, and task-specific design understanding. These features are inspired by published theories of human cognition during design tasks. We have discussed the features in the context of Argo/UML, however, we believe them to be useful in many design contexts and tools.

Our immediate research plans include the addition of new cognitive features to Argo/UML and evaluation of the impact of these features. We have already conducted one user study of the broom alignment tool [18], and plan to conduct two more studies of navigational perspectives and design critics.

Our overall goal in developing Argo/UML is to test how our cognitive features can add value to a production quality CASE tool. We have realized that goal to a large extent and also produced a free, open-source CASE tool that has been used in academic and industrial settings. Furthermore, we encourage other researchers to use Argo/UML as a starting point and test-bed for their own features.

References

1. Al-Shamma, N., et al. Precision Graphics Markup Language (PGML): World Wide Web Consortium Note 10-April-1998. Available from http://www.w3.org/TR/1998/NOTE-PGML.
2. Beck, K. and Johnson, R. Patterns generate architectures. Proc. European Conf. on Object-Oriented Programming (ECOOP'94). Bologna, Italy. 1994.
3. Bonnardel, N. and Sumner, T. Supporting evaluation in design: the impact of critiquing systems on designers of different skill levels. Acta Psychologica. vol. 91. 1996. pp. 221-244.
4. Chun, H. W. and Lai, E.M.-K. Intelligent critic system for architectural design. Trans. Knowledge and Data Engineering. July 1997.
5. Fischer, G., Girgensohn, A., Nakakoji, K., Redmiles, D. Supporting Software Designers with Integrated, Domain-Oriented Design Environments, IEEE Transaction on Software Engineering, Special Issue: "Knowledge Representation and Reasoning in Software Engineering", vol. 18, no. 6, pp. 511-522.
6. Fu. M. C., Hayes, C. C., and East, E. W. SEDAR: expert critiquing system for flat and low-slope roof design and review. Journal of Computing in Civil Engineering. vol. 11. no. 1. January 1997. pp. 60-68.
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. 1994.
8. Gertner A. S. and Webber B. L., TraumaTIQ: online decision support for trauma management. IEEE Intelligent Systems. January/February 1998. pp. 32-39.
9. Green, T. R. G. and Petre, M. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. J. Visual Languages and Computing, vol. 7, no. 2 (June 1996). pp. 131-174.
10. JavaBeans Specification 1.01. Available from http://java.sun.com/beans/.
11. Kintsch, W. and Greeno, J. G. Understanding and Solving Word Arithmetic Problems. Psychological Rev. vol. 92. 1985. pp. 109-129.
12. Langlotz, C. P. and Shortliffe, E. H. Adapting a consultation system to critique user plans. International Journal of Man-Machine Studies. vol. 19. no. 5. Nov. 1983. pp. 479-496.
13. Lemke, A. C. and Fischer, G. A cooperative problem solving system for user interface design. AAAI-90. 1990. pp. 219-240.
14. Lowgren, J. and Nordqvist, T. Knowledge-based evaluation as design support for graphical user interfaces. Conference Proceedings on Human Factors in Computing Systems (CHI'92). 1992. pp. 181-188.
15. Object constraint language specification. Object Management Group document ad/97-08-08. Sept. 1997. Available from http://www.omg.org/docs/ad/97-08-08.pdf.
16. Pennington, N. Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. Cognitive Psychology. vol. 19. 1987. pp. 295-341.
17. Riel, A. Object-Oriented Design Heuristics. Addison-Wesley: Reading, MA. 1996.
18. Robbins, J. E., Kantor, M. and Redmiles, D. F. Sweeping away disorder with the broom alignment tool. Conference Proceedings on Human Factors in Computing Systems (CHI'99). May 1999.
19. Schoen, D. The Reflective Practitioner: How Professionals Think in Action. 1983. New York: Basic Books.
20. Silverman, B. G. and Mezher, T. M. Expert critics in Engineering design: Lessons learned and research needs. AI Magazine, Spring 1992.
21. UML Revision Task Force. UML v1.3 R20 Metamodel abstract syntax. Jan. 1999. Available from http://uml.shl.com
22. UML Semantics. Object Management Group document ad/97-08-04. Sept. 1997. Available from http://www.omg.org/docs/ad/97-08-04.pdf.
23. Visser, W. More or Less Following a Plan During Design: Opportunistic Deviations in Specification. Int. J. Man-Machine Studies. 1990. pp. 247-278.
24. Vlissides, J. M., Linton, M.A. Unidraw: A framework for building domain-specific graphical editors. ACM Transactions on Information Systems, vol. 8, no. 3, July 1990. pp. 237-268.
25. XML Metadata Interchange (XMI). Object Management Group document ad/98-07-01. July. 1998. Available from http://www.omg.org/docs/ad/98-07-01.pdf.