CHAPTER 1: Introduction

1.1 Background on CASE tools

What are CASE tools. *Computer Aided Software Engineering* (CASE) is a catchall term to describe virtually any software development tool. However, it is typically used to refer to software development tools that use diagrammatic representations of software design. For the purposes of this dissertation, the term is further specialized to mean tools that support the use of design diagrams in the development of object-oriented software. These tools are also known as OOAD (Object-Oriented Analysis and Design) tools.

CASE history in a nutshell. 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 tools served as electronic replacements for paper, pencil, and stencil. Many of these early CASE tools became unused “shelfware” because they did not provide significant value to software designers (Iivari, 1996). 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.

International Data Corporation (IDC), a market research firm that collects data on all aspects of the computer hardware and software industries, has published a series of

reports on OOAD tools. “IDC expects revenues in the worldwide market for OOAD tools to expand at a compound annual growth rate of 54.6% from $127.4 million in 1995 to $1,125.2 million in the year 2000” (IDC 1996). Much of this growth is expected to stem from adoption of OOAD tools by smaller software development organizations.

The need to help designers. Software design is not simply an automateable 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 their 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 (Fox, 1997; PITAC, 1999)

1.2 Research Method

Sophisticated development tools such as CASE tools have dozens or hundreds of features. Two common ways that tool builders decide which features to include are by looking at the features that similar tools provide and by directly observing users. Direct observation of users is followed by interpretation of the observations, generation of proposed features, and evaluation. The basic difference in my approach is that I generate

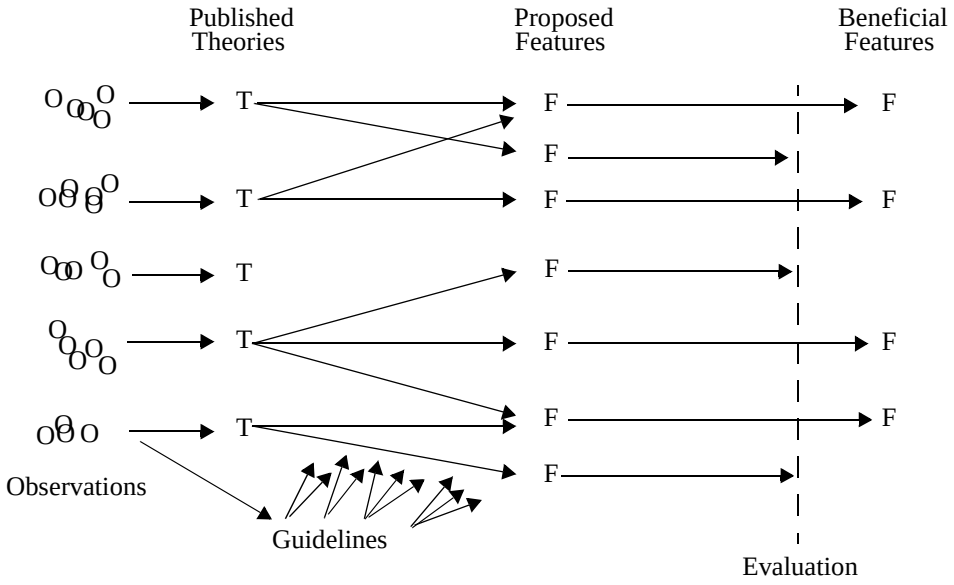

Figure 1-1. Freature generation approach

proposed features based on published theories of designers' cognitive needs. These theories are themselves based on careful observations of human activity. The main advantage of using published theories stems from their insights into human design activities, which are deeper and more widely applicable than those a typical CASE tool builder can glean from direct observation. My approach is illustrated in Figure 1-1.

This dissertation presents a set of novel design tool features intended to support designers in making and ordering decisions. Each of these features is motivated by experience in designing software systems and by published theories of the cognitive challenges of design. Many of these theories have been published in books and journals on cognitive psychology while others are implicit in the software engineering literature. Using cognitive theories to guide the development of design tool features has resulted in several promising new features that I would not otherwise have been likely to invent.

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. It also includes enough standard CASE tool functionality to be generally useful. Argo/UML is implemented in Java and consists of over 100,000 lines of code in over 800 classes. Preliminary versions of Argo/UML have been 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 classrooms. Argo/UML is the fourth in a series of tools that I have developed or enhanced with cognitive support features. Although Argo/UML is the focus of this dissertation, the design support features described are also applicable to other design tools. This point will be further discussed in the conclusion.

A good research method should be repeatable. I expect the research method described in Figure 1-1 to be repeatable by researchers with background similar to my own. Specifically, the method requires researchers to become familiar with several theories of designers' cognitive needs, to be experienced users of existing design tools in the target domain, and to have a broad knowledge of standard usability issues. One of the contributions of this dissertation is a description of the relevant theories in a way that could allow more people to repeat the method.

Advanced user interface features cannot make an unusable design tool into a usable one. Current CASE tools certainly leave room for improvement in basic usability. For example, they often make poor use of limited screen space, confuse users with a dizzying array of commands, require excessive use of the mouse rather than the keyboard, provide commands that are not well matched to the way designers think about the design task, and

impose unneeded modality. I have made some efforts to improve the basic usability of Argo/UML, and these efforts are also grounded in theory and Human-Computer Interface (HCI) guidelines.

1.3 Hypothesis and Contributions

The hypothesis of this dissertation is that cognitive theories of design can inform the development of CASE tool features that provide cognitive support to designers. Here "CASE tool" refers to object-oriented software design tools. Implications for other types of software development tools are discussed in the final chapter. Furthermore, "inform" means that parts of the theories described in Chapter 2 lead to parts of the features proposed in Chapter 4. Finally, "provide cognitive support" means that each feature's expected benefits can be explained in terms of the theories of designers' cognitive needs and that the feature has a positive effect on the designers' mental processes or work products.

The contributions of this dissertation are as follows:

- It describes theories of design cognition in terms understandable and relevant to CASE tool builders (see Chapter 2).
- It proposes a basket of useful CASE tool features. Each feature is inspired and explained by cognitive theories and HCI guidelines (see Chapter 4). Several of these features are novel, while others are improvements on features that may be found in current CASE tools or research systems.
- It demonstrates the successful application of a theory-based UI design method to a large-scale software engineering tool.
- It describes the design of a scalable, reusable infrastructure for building the proposed features in design tools (see Chapter 8).

1.4 Organization of the Dissertation

This dissertation consists of nine chapters. The next two chapters present cognitive theories of design and previous work on design critiquing systems. Chapter 4, the heart of the dissertation, proposes cognitive support features that address some of the needs identified in the cognitive theories. Chapter 5 demonstrates how these features might be used. Chapter 6 evaluates the proposed cognitive support features using heuristic usability methods, while Chapter 7 evaluates the features and approach empirically. Chapter 8 describes the reusable libraries that implement the cognitive support features. Chapter 9 concludes the dissertation.

CHAPTER 2: Theories of Designers' Cognitive Needs

This chapter presents the cognitive theories that informed the design of cognitive support features in Argo/UML. As mentioned in Chapter 1, the majority of these theories are drawn from cognitive science journals and books. However, some are drawn from software engineering or human-computer interaction literature.

Many of the theories described in this chapter propose a model of human thought, memory, or activity during problem solving and design. These models are based on lower level cognitive models, observations, and controlled experiments. However, they are simply models: they are consistent with observations but they do not claim to identify the root or sole cause of observed behavior. Although many of these theories are widely accepted, none of them are irrefutable. As in any active research field, new cognitive models and theories will periodically be introduced. In fact, several of the subsections in this chapter include references to articles questioning the theories presented. Furthermore, since I am primarily a software engineer rather than a cognitive psychologist, my mastery of these cognitive theories has its limits. My research method reduces the risk of depending on an incorrect, inappropriate, or misunderstood theory by drawing from multiple cognitive theories, multiple user interface guidelines, and the experience of others and myself in using similar tools.

In the following subsections, each cognitive theory is described in terms accessible to software engineering tool builders, and the implications of the theory on tool building are outlined. The implications can be classified into two broad levels: first, design tools

should avoid interfering with the natural thought processes of designers; and, second, design tools should actively support these processes when possible.

2.1 Theories of Design Decision-Making

2.1.1 Reflection-In-Action

Description. The cognitive theory of reflection-in-action (Schoen 1983, 1992) observes that designers of complex systems do not conceive a design fully-formed. 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 particular machine resources. That decision is usually made in the context of other decisions about inter-component communication and machine resource allocation. 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.

One interesting related effect is "the tyranny of the blank page" (Boucher, 1995). Authors (who are a type of designer) often find it hard to start from nothing. Starting from a partially specified design (e.g., a rough outline) is much easier, even if much of this initial template is changed in the course of design. This may be because it is easier to make design decisions in *some* context of decisions that have already been made, rather than making a *first* decision in a context consisting only of future decisions. This is partly due to the fact that many design problems are underconstrained and the designer must add

constraints or assumptions to properly frame the problem. Here, however, I am more concerned with the way that a design document plays the role of an external memory that augments the designer's short and long-term memory.

The need to make decisions in the context of design may be partly explained in terms of more basic theories of human associative memory structure and access (e.g., Cofer, 1975). People store their memories in mental structures and use the structures to retrieve memories. Some parts of the structure are more active at a given time, based on recent accesses. Designers may possess specific design knowledge, but they will not be able to recall and apply it unless they can activate the proper section of their memory structure. When engaged in the design process, designers can use aspects of the partially specified design as prompts to activate needed memories. For example, when adding a new class to an object-oriented design a designer may visually scan the other classes already in the design to decide what new relationships are needed; each existing class icon serves as a prompt to activate the designer's memory of relevant domain semantics. More generally, when designers look at their design and ask questions such as "What is missing *here*?" or "What is wrong with *this* part?" they are using information in the external design document as cues to activate needed parts of their associative memory. Section 2.2 discusses human memory structures in more detail.

The "reflection" in reflection-in-action can include any type of analysis that raises design issues. While visual inspection of the design is one of the most frequent types of analysis, domain-specific analyses can provide more specific cues. Knowing which questions to ask about a design is part of a designer's expertise. Making use of that

expertise depends on being able to recall them from the designer's associative memory at the time when they are needed. This leads to a cycle of synthesis and analysis: each design decision prompts the designer to apply the appropriate analyses, and each analysis produces new prompts for the designer to consider new design decisions.

Implications for design support. The theory of reflection-in-action implies several requirements for design tools. First, the designer should be allowed to easily alternate between synthesis and analysis activities. The designer should not be forced to complete a synthesis activity without analysis; likewise, the designer should not be locked into an analysis mode. Furthermore, the tool should support the natural integration of synthesis and analysis by providing the designer with prompts to recall relevant design knowledge. Many of the cognitive support features in Argo/UML are inspired by this theory, including design critics, checklists, non-modal wizards, the dynamic "to do" list, clarifiers, and the create multiple feature.

2.1.2 Opportunistic Design

Description. 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 lack 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. Here, searching for information can occur at two levels: physically searching for books, web pages, or knowledgeable colleagues, or mentally searching for elements of the designer's knowledge that are not immediately accessible. Complexity of a task roughly corresponds to the number of smaller tasks that comprise it.

On the other hand, opportunistic switching can occur when one task brings to mind the information needed for another task. In these situations, the designer may defer completion of the original task and embark on a design excursion to address the second task before the needed information slips out of mind (looses activation). The designer should eventually return from each excursion to complete the original task, but the need to remember to return imposes a short-term memory load.

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.

Implications for design support. 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 tools can allow the benefits of both an opportunistic and a prescribed design process. They should allow, and where possible augment, human designers’ abilities to choose the next design task to be performed. Furthermore, design tools should aid designers in returning to the prescribed design process after they complete each opportunistic excursion. Process support should exhibit the following characteristics to accommodate the opportunistic design process.

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. As discussed below, high loads on short-term memory can induce procedural errors by reallocating short-term memory slots that were used to keep track of pending steps or goals; reminding provides designers with an external and reliable memory for pending steps and goals.

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.

The theory of opportunistic design has influenced many of Argo/UML's features. The dynamic "to do" list, checklists, opportunistic table views, and opportunistic search utility are the features most heavily influenced by this theory.

2.1.3 Geneplore

Description. Finke, Ward, and Smith (1992) present a model of creative design activity called Geneplore. Geneplore consists of two cyclically recurring phases: generation and exploration. During the generation phase new ideas are generated but not evaluated. During the exploration phase the ideas are explored and their implications are evaluated.

This theory is akin to reflection-in-action, in that it identifies synthesis activities as distinct from analysis activities. However, reflection-in-action emphasizes the complementary nature of these two activities, whereas Geneplore warns of possible interference.

Many practical creative techniques are based on separating synthesis from analysis to reduce interference. For example, Osborn (1953) developed a popular brainstorming technique that involves listing and free association of ideas in a group setting where

participants must withhold criticism until later. Also, Austin (1994) offers four exercises to break out of writers' block: write sentences using words selected randomly from a dictionary, build a dialog around one line taken out of some other context, write an informal letter to a friend about the task, or write vignette about family pictures. These exercises help break writers' block by encouraging the writer to focus on the generation phase of a task that has no inherent evaluation and then return to the original writing task to generate new ideas.

Finke, Ward, and Smith (1992) review several proposed techniques for generation of new ideas during the generation phase, including attribute listing, defining a space of possible attribute values, and visual combination.

Implications for design support. The theory of reflection-in-action implies that design tools should simultaneously support both synthesis and analysis. Geneplore, on the other hand, implies that design support tools can encourage creativity by helping designers separate synthesis and analysis. Furthermore, brainstorming and other idea generation techniques have been found to be useful in many fields, but they are usually manual and rarely directly supported by design tools.

Argo/UML's visual blender feature is directly inspired by this cognitive theory.

2.2 Theories of Human Memory

2.2.1 Associative Recall

Description. Cognitive scientists use many alternative models of human semantic memory. One aspect found in virtually all of these models is that of associative recall

(e.g., Cofer, 1975; Ellis and Hunt, 1993). Basically, associative recall allows people to recall related concepts when they are presented with cues that activate one part of their memory. For example, people can more easily recall a list of meaningfully related words than a list of random ones. Collins and Loftus (1975) proposed the Spreading-Activation Model of human memory. In this model, concepts are connected to others in a network with links of various strengths. Activation of one memory node can spread to linked nodes, causing them to become somewhat activated as well. Several studies have indicated that spreading activation can help explain performance on free recall of word lists (e.g., Ross and Bower, 1981).

However, others have cast doubt on the applicability of free recall performance to decision-making and design processes. Anderson (1996) suggests that subjects often combine concepts into exemplars and then make decisions based on the exemplars, even if they cannot recall the original concepts. This is just one example of the type of mental construction that may take place when accessing human memory. Theories of cognitive fixation also apply to design and are discussed below.

Implications for design support. This theory implies that design tools can present cues that cause designers to retrieve their own related memories. For example, effective cuing could allow a knowledge-poor tool to aid knowledge-rich users in applying their knowledge when needed. On the other hand, off-topic cues can also distract users, causing them to think of things not related to the task at hand.

These cues can be overtly generated by the design tool as help files, examples, error messages, or design critiques. However, the visible state of the design is itself a strong cue that is always present and frequently used.

Nearly every cognitive support feature in Argo/UML makes use of this theory of associative recall. The clarifier, opportunistic search, create multiple, and visual blender features relate most directly to this theory. Many user interface guidelines are also derived from this underlying aspect of human memory.

2.2.2 Limited Short-Term Memory

Description. Another aspect found in virtually all models of human memory is short-term memory (STM). Miller (1965) first proposed that STM has a capacity of seven plus or minus two items. Using a simple number ignores the effects of chunking, primacy, and recency. Chunking occurs when people group related items based on knowledge from their long-term memory, so short-term and long-term memory are obviously linked. Primacy and recency effects cause people to remember the first and last words in a given list more easily than those in the middle.

The general idea is that designers need to focus on a subset of the design and a subset of design issues. Minor distractions, such as working through a complex user interface, can take up slots in a designer’s short-term memory and knock out application specific facts. Byrne and Bovair (1997) conducted an experiment that showed that increased short-term memory loads caused procedural errors in carrying out complex user interface tasks. Specifically, subjects with high STM loads committed super-goal kill-off errors, i.e., they thought that they had finished the task when they had really only finished a

major step. Furthermore, in discussing comprehension and problem solving of word arithmetic problems, Kintsch and Greeno (1985) cite an earlier study (Kintsch and Polson, 1979) that showed that "there may be trade-offs between the capacity of the short-term [memory] and the other resource demands on the [person], so that if the task to be performed is a difficult one, fewer resources are available for actively maintaining information in the [memory]."

Implications for design support. One implication of this theory is that limited short-term memory resources are used in both making design decisions and in forming plans for tool usage. This suggests that a design tool user interface should reduce short-term memory load where possible. In fact, this is a general user interface design guideline that has also been widely applied to other types of user interfaces (e.g., Shneiderman, 1998). Another way that design tools can support designers is by augmenting their short-term memory with external memory provided by the tool. This might be as explicit as prompting the designer to enter textual notes on what he or she wants to remember, or it may be as implicit as placing the cursor in the proper field to prompt the designer to continue an interrupted task at the point where it was left off.

Limited short-term memory resources are also used to mentally combine views offered by the tool into task-specific mental models. Tools might help reduce these short-term memory demands by providing views that better match the designer's task-specific mental models or by aiding the designer in visualizing combinations of views.

As with associative recall, many user interface guidelines take into account the limited short-term memory of tool users. Some of Argo/UML's cognitive support features

aim to reduce short-term memory loads by simplifying interactions with the tool or providing reminders. Specifically, Argo/UML's selection-action buttons, create multiple feature, and broom alignment tool each help simplify the interactions needed for common design tasks. Also, Argo/UML's clarifiers, dynamic "to do" list, checklists, and design history help remind the designer of information that they might otherwise forget.

2.2.3 Cognitive Fixation

Description. Cognitive fixation occurs when memory cues block retrieval of related concepts rather than aiding their retrieval. For example, in one experiment, when subjects were asked to memorize a list of random words and then cued with some of the same words, retrieval of the other words was actually lower than when no cues were provided. This effect is explained by Smith, Ward, and Schumacher (1993) as memory retrieval interference between activation of the cue and of the other words. Several experiments have shown that providing subjects with examples can reduce novelty in creative tasks, even when subjects explicitly try not to follow the example. For instance, Jansson and Smith (1991) asked engineering students and professional designers to design a measuring cup for blind users. They found that subjects tended to fixate on provided examples, even if those examples lacked needed features. Smith, Ward, and Schumacher (1993) found that subjects fixated on examples just as much when they were explicitly instructed to make their solution different from the example as when they were given no such instructions. Furthermore, Ward (1994) found that students and professional writers can often become fixated on self-generated cues and initial assumptions.

Feeling-of-knowing and tip-of-tongue mental states can also be caused by fixation when an incorrect word prevents retrieval of the correct word. Smith has conducted several cognitive studies that have induced tip-of-tongue states and investigated the impact of incubation on these states (Smith, 1994). Here, the term “incubation” refers to a period of idle time between the initial retrieval failure and subsequent successful retrieval. During this time delay, subjects are occupied with unrelated tasks. In one study, a twenty-three minute incubation had no significant reduction of fixation (Smith and Vela, 1991); in contrast, another study found that one minute of incubation substantially reduced fixation (Meyer and Bock, 1992).

Design novelty and productivity can be reduced when designers fixate on one alternative to such an extent that it interferes with the generation of other alternatives. This can occur when attempts to think of an alternative to a design fragment results in reactivation of the same parts of the designer’s memory that lead to the current fragment.

Implications for design support. Design support tools might address fixation on initial assumptions by prompting designers to consider alternatives suggested by the tool. However, these prompts may distract the designer, possibly inducing fixation themselves. On the other hand, short distractions can be useful if they break a fixation activation cycle by providing an incubation period.

Fixation has not inspired a cognitive support feature directly, but it has been used to explain why some of the proposed features are expected to succeed or fail. Specifically, design critics, checklists, create multiple, and the visual blender feature all take fixation

into account. The theory of fixation, much like the theory of opportunistic design, has helped me avoid proposing features that are inconsistent with the theory.

2.2.4 Limited Knowledge

Description. In 1987, Guindon, Krasner, and Curtis (1987) identified several difficulties faced by software designers:

> 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 postponed; and (7) difficulty in expanding or merging solutions from individual subproblems to form a complete solution. (Guindon, Krasner, and Curtis, 1987)

Several of these difficulties are addressed by other cognitive theories described in this chapter. For example, difficulty in keeping track of postponed subproblems is discussed along with the theory of opportunistic design, and difficulty in considering constraints is addressed in the discussion of reflection-in-action. This subsection focuses on the first two difficulties: lack of domain knowledge and lack of process knowledge.

The next year, Curtis, Krasner, and Iscoe (1988) elaborated on the “thin spread of application domain knowledge” as one of the main difficulties of developing large software systems. This thin spread is another way of saying that no single designer knows everything that he or she will need to know to complete a complex design. Even an expert in a narrow domain will have to step outside that domain to achieve a complete design. Also, leading-edge or even market-competitive designs usually push the boundaries of

what is well known. Because of the thin spread of knowledge, designers need to interact with other stakeholders on any non-trivial design project to access the knowledge that those people hold.

Implications for design support. The simple fact that no one knows everything implies that design support tools can help by providing knowledge that designers lack. This knowledge may take the form of explanations, rules, examples, templates, or suggestions. If the tool does not have built-in knowledge in a given area, it may offer a suggestion as to where that knowledge may be found or who has it. Critics and checklists are two tool features that can deliver this knowledge.

In addition to the complexity of the application domain, design tools are themselves complex and designers require substantial tool-specific knowledge to use them. Even if a given tool is used frequently, it will likely contain features that are rarely used by any given designer. This suggests that user interface guidelines intended to support occasional and incremental learning should be applied to high functionality design tools (Fischer 1989).

2.2.5 Mental Biases

Description. Stacy and MacMillian (1995) argue that people are frequently biased irrationally by their experience, particularly in regard to the weight that they give to one experience over another. Representativeness biases cause people to generalize too quickly from a particular experience to the class of experiences, or assign aggregate properties of the class to individual instances. Availability biases favor ideas that are easily brought to mind or visible over those that are harder to recall or invisible. Confirmatory biases favor

experiences that confirm one's hypotheses over experiences that disprove it. Mental models that are easier to think about tend to be used more often than ones that are harder to think about, even if the more difficult models are more appropriate to the decision at hand.

Implications for design support. Design support tools can help by counter-acting some of these biases. For example, if designers are biased in favor of analyses that confirm the quality of their design, the design tool should use features such as critics or checklists to prompt them to consider analyses that question that quality. Likewise, easily invoked computer simulations could help reduce designers' reliance on mental simulation of simplified mental models.

2.3 Design Visualization Theories

2.3.1 Comprehension and Problem Solving

Description. 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 development, 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). Specifically, multiple representations such as program syntactic decomposition, state transitions, control flow, and data flow enabled programmers to better identify elements and relationships in the problem and solution and thus more readily 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).

Implications for design support. 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 elements may appear in multiple perspectives. Coordination among design perspectives ensures that elements 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, 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 can help support communication among stakeholders with diverse backgrounds and mental models. Such communication is key to developing complex systems that are robust and useful.

It is my contention that no fixed set of perspectives is appropriate for every possible design; instead perspective views should emphasize what is currently important to the

designer. When new issues arise in the design, it may be appropriate to use a new perspective on the design to address them. While I 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).

2.3.2 Secondary Notation

Description. Design diagrams in any given field follow a formal syntax that assigns meaning to specific graphical shapes, connections, and labels. Beyond formal syntax, a set of diagramming conventions exists within each design community. However, there are other visual aspects of diagrams that are left to the designer's discretion. These may include the color, size, location, spacing, and alignment of diagram elements. Designers use these unassigned visual aspects in a "secondary notation" that expresses relationships that are of concern, but that are not covered by the formal notation (Green and Petre, 1996).

Programmers using conventional programming languages express their program to the compiler through formal syntax, but they also use naming conventions, indentation, and blank lines to aid communication with other programmers. These informal structures form a "secondary notation" that can aid the reader in identifying commonalities between different program elements and in breaking down large structures into understandable chunks. Likewise, in a design diagram, objects can be aligned to show logical structure, grouping, correspondence, or emphasis.

Green and Petre (1996) point out that visual similarity between two design fragments can define a visual "rhyme" that cues the reader to expect deeper semantic

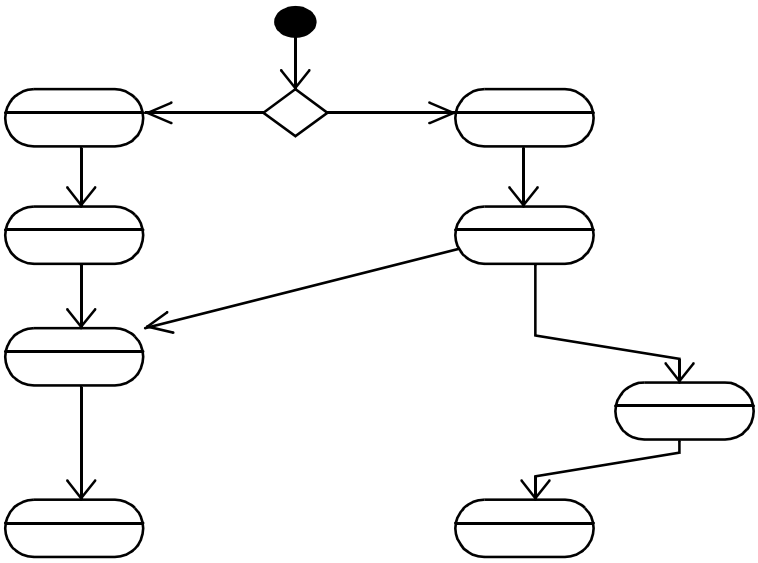

Figure 2-1. State diagram with alignment as secondary notation

correspondence. For example, in Figure 2-1, alignment implies grouping and correspondence of steps. Formally, the diagram consists of a single set of undifferentiated states. But designers experienced in using this type of diagram should perceive two sets of states with corresponding parts.

Implications for design support. The observation that designers use secondary notation identifies an important usability requirement for design diagram editors. Specifically, design editors should actively help designers in maintaining elements of secondary notation. However, the interfaces of most of these tools are inspired by generic drawing tools and do not provide specific support for secondary notation. In contrast, programmers' text editors such as emacs (Stallman, 1979) do actively support indentation and alignment as secondary notation. Argo/UML's broom alignment tool and selection-action buttons help achieve secondary notation in UML diagrams.

2.3.3 Viewing as an Acquired Skill

Description. Diagrammatic design representations are not necessarily understandable to anyone at first glance. Diagrams have a certain immediacy of communication, but only to those who have experience in reading that type of diagram. Due to the thin spread of application knowledge, not all stakeholders can be assumed to have the skills needed to read all diagrams produced in a given project or to produce diagrams that fit the norms of the design community (Petre, 1995).

Implications for design support. One implication of Petre's observation is that design diagramming tools should take the norms of the design community into account. For example, an object-oriented design tool should follow the norm of laying out subclasses below their superclasses. In Argo/UML, selection-action buttons encourage the designer to follow conventions during diagram construction. Another implication is that some designers may need support in understanding diagramming conventions. For example, a legend or link to on-line help might aid designers who are new to the domain.

2.4 User Interface Guidelines

2.4.1 Style Guidelines and Usability Heuristics

Many different types of user interface guidelines have been proposed. Some of these are specific rules that define a given window system look and feel, others are more

general-purpose heuristics. Tables 2-1 through 2-5 present excerpts from some well-known sets of guidelines and heuristics.

Table 2-1: Usability guidelines from Mac Look and Feel (Apple 1993)

Metaphors: Use metaphors involving concrete, familiar ideas.
Direct Manipulation: Allow people to feel that they are directly controlling the objects represented by the computer.
See-and-Point: Users interact directly with the screen, selecting objects and performing activities by using a pointing device.
Consistency: Use the standard elements of the Macintosh interface to ensure consistency within your application and to benefit from consistency across applications.
WYSIWYG (What You See Is What You Get): Make sure that there is no significant difference between what the user sees on the screen and what the user receives after printing.
User Control: Allow the user, not the computer, to initiate and control actions.
Feedback and Dialog: When a user initiates an action, provide some indicator that your application has received the user's input and is operating on it.
Forgiveness: People need to feel that they can try things without damaging the system.
Perceived Stability: Provide a clear, finite set of objects and a clear, finite set of actions to perform on those objects. When actions are unavailable, they are not eliminated but are merely dimmed.
Aesthetic Integrity: Design your products to be pleasant to look at on the screen for a long time.
Modelessness: Allow people to do whatever they want when they want to in your application.

Table 2-2: Usability guidelines from Java Look and Feel (Sun, 1999)

The most effective method of laying out user interface elements is to use a design grid with blank space to set apart logically related sets of components.
Use headline capitalization for most names, titles, labels, and short text. Use sentence capitalization for lengthy text messages.
Specify keyboard shortcuts for frequently used menu items to provide an alternative to mouse operation. Be aware of and use common shortcuts across platforms.
Avoid the use of a second level of sub-menus. If you want to present a large or complex set of choices, display them in a dialog box.

Table 2-3: Usability guidelines from Nielsen (1995)

Visibility of system status
Match between system and the real world
User control and freedom
Consistency and standards
Error prevention
Recognition rather than recall
Flexibility and efficiency of use
Aesthetic and nominalist design
Help users recognize, diagnose, and recover from errors
Help and documentation

Table 2-4: Usability guidelines from Constantine and Lockwood (1999)

Structure: put related things together and separate unrelated things
Simplicity: make common tasks simple
Visibility: keep all options and materials for a given task visible
Feedback: keep users informed of actions, state, and errors
Tolerance: accept reasonable inputs, reduce the cost of errors
Reuse: do similar things in similar ways

Table 2-5: Usability guidelines from Shneiderman (1998)

Recognize human diversity
Strive for consistency
Enable frequent users to use shortcuts
Offer informative feedback
Design dialogs to yield closure
Offer error prevention and simple error handling
Permit easy reversal of actions
Support internal locus of control
Reduce short-term memory load

Several of these heuristic guidelines coincide with the cognitive theories of design. For example, Nielsen's heuristic that recognition is better than recall can be explained by the cognitive theory of associative memory: specifically, users will find it easier to recall an abstract concept when they are presented with a visual cue that they associate with that concept.

Style guidelines and usability heuristics may have a basis in experience and underlying cognitive theories, but they are stated with an emphasis on ease of technology transfer and application. Nielsen motivates his set of ten usability heuristics with the comment that “even the best method will have zero impact on the product if it does not get used” (Nielsen, 1993). Non-expert interface designers are likely to achieve better results more easily when following established guidelines than when inventing new interactions without guidance. However, general-purpose guidelines and heuristics have not by themselves resulted in very satisfactory CASE tool user interfaces. For example, Rational Rose adheres to the MS Windows user interface guidelines fairly closely and it imitates some of the dialog boxes found in other popular development tools, yet the overall usability of Rose is low due to its modality and lack of specific support for common design tasks.

2.4.2 Fitts’ Law

Description. Fitts’ Law addresses the low-level physical and cognitive task of indicating a position in a two-dimensional space. Briefly stated, the time required for one to move one’s hand (or mouse) from a starting region to a target region depends on the distance moved and the size of the target region (Fitts, 1954). Moving to a distant target takes longer because of the distance traveled, and moving to a small region takes more time because of the need for precise positioning.

Implications for design support. This theory calls into question any user interfaces that requires the user to move the mouse to small and distant target areas. Yet, many examples of such interfaces are found in daily use. For example, a standard scrollbar has

very small and distant target areas to scroll up and scroll down. The difficulty of using such an interface goes unnoticed by most users until they are exposed to an easier alternative. For example, the “wheel mouse” sold by Microsoft and other companies allows users to scroll without changing the mouse position.

Software design tools typically focus on design diagrams that contain both graphical elements and structured text. Editing of these diagrams entails substantial switching between mouse and keyboard input devices. Furthermore, many current CASE tools use complex, modal dialogs that force designers to interact with small widgets. All this can add up to substantial arm and wrist stress and a perception of difficulty in using the tool.

Since design tools are used daily by designers, the cumulative effects of physical stress and perceived difficulty can be substantial. Design tools should reduce the effort needed for common operations. Knowing which operations are most common and in which contexts they will be performed requires design tool builders to use their knowledge of domain-specific design tasks. For example, selection-action buttons in Argo/UML aid designers in constructing common types of nodes and edges in design diagrams.

CHAPTER 3: Previous Work in Cognitive Features for Design Tools

3.1 Previous Work on Design Critiquing Systems

3.1.1 Definitions of Design Critiquing Systems

A design critic is an intelligent user interface mechanism embedded in a design tool that analyzes a design in the context of decision-making and provides feedback to help the designer improve the design. Feedback from critics may report design errors, point out incompleteness, suggest alternatives, or offer heuristic advice. One important distinction between critics and traditional analysis tools is the tight integration of design critics into the designer's task: critics interact with designers while they are engaged in making design decisions.

Table 3-1: Selected sefinitions of critiquing systems

Langlotz and Shortliffe (1983) describing ONCOCIN: "A critique is an *explanation of the significant differences* between the plan that would have been proposed by the expert system and the plan proposed by the user."
Miller (1983) on ATTENDING: "A critiquing system is a computer program that *critiques human generated solutions*."
Fischer et al. (1991) on Janus: "Critics *operationalize Schoen's* concept of a situation that talks back. They use knowledge of design principles to detect and critique suboptimal solutions constructed by the designer."
Sumner, Bonnardel, and Kallak (1997) describing VDDE: "Critiquing systems embedded in [design] environments augment designers' cognitive processes by analyzing design solutions for *compliance with criteria and constraints* encoded in the system's knowledge-base."

Table 3-1 shows some of the definitions of critiquing systems found in the literature. I have added italics to each definition to highlight key phrases that differentiate it from the others.

The definition in Table 3-1 given by Langlotz and Shortliffe defines critiques as explanations of differences. Their system, ONCOCIN, arose from an effort to increase the explanation producing power of an existing expert system. The emphasis was on the system's solution; the doctor's solution was used only to choose which parts of the system's solution needed to be explained. The hope was that better explanation capabilities would make the system more acceptable to its users.

Miller's definition of critiquing system places more emphasis on the user's solution. Miller's system, ATTENDING, was developed in an effort to make medical consulting expert systems more acceptable to their intended users, much like ONCOCIN.

The first two definitions in Table 3-1 are early ones that do not imply much interaction between the designer and the system. In contrast, the definition given by Fischer and colleagues introduces a cognitive aspect that shifts the primary focus away from simple observations of user acceptance and to the cognitive needs of human designers. Support for Schoen's theory of reflection-in-action implies a tight integration of critics into design tools and a significant level of interaction between designers and critics during design tasks. It is this definition of critiquing that is closest to my own.

The last definition is representative of much of the more recent work in critiquing that consists of the application of the critiquing approach to new domains. It speaks of applying arbitrary criteria and constraints, and critiquing is viewed as a user interface approach that is distinct from the underlying knowledge-base.

My definition of critiquing differs from those in Table 3-1 in several ways. I position critiquing as an intelligent user interface mechanism that can add value to standard direct-manipulation or forms-based design tools, rather than as a more acceptable repackaging of expert system technology. Like Fischer's definition, I require that critics provide cognitive support for human decision-making, but I do not limit that support to a single theory of design. All of the definitions in Table 3-1 stop at informing the designer of the existence of problems; I go a step farther by defining the goal of critiquing as helping to carry out design improvements. I use the term "constructive" to emphasize that a critic provides this additional level of support.

Definition. A design critic is an intelligent user interface mechanism embedded in a design tool that analyzes a design in the context of decision-making and provides feedback to help the designer improve the design.

A critiquing system includes more than merely critics. A critiquing system must support the application of critics during design. However, most also include support for critic authoring, management of the feedback from critics, or a strategy for scheduling the application of critics.

3.1.2 Previous Work on Critiquing Processes

The definitions of critiquing systems given in Table 3-1 imply a simple detect-advise process: (1) critics detect potential problems in a design, and (2) these critics advise the designer of the problems. Critiquing systems can be evaluated based on their support for these two phases, but they must also be evaluated with respect to the relevance of their

Figure 3-1. Phases of the ADAIR critiquing process

design feedback to the designer's current task, and support for guiding or making design improvements.

Some previous research efforts have extended the detect-advise critiquing process. The Janus family of critiquing systems adds a new phase to the beginning of the detect-advise critiquing process: appropriate critics are activated based on a specification of design goals. The TraumaTIQ system, like Janus, activates critics based on design goals; however, in TraumaTIQ goals are inferred from the user's actions rather than stated directly. Sumner, Bonnardel, and Kallak (1997) define a critiquing process with three major steps: analyzing the design, signaling design errors, and delivering rationale that explains the problem and possible solutions. In addition to the phases of the detect-advise process, this process outlines the improvement activities of the designer. As described below, I have attempted to merge and extend these process models to clarify the role of critics and document the functionality of the Argo critiquing system. The resulting process model is described below.

3.1.3 Phases of the ADAIR Process

The ADAIR critiquing process is named after the five phases that make up the process: Activate, Detect, Advise, Improve, and Record. Design support systems and designers repeatedly work through these phases over the course of a design. The phases are shown in Figure 3-1 as a linear sequence, however, some phases may be skipped in

certain situations, and multiple instances of the process may be concurrently active at any given time. The ADAIR phases are not necessarily contiguous: other work often intervenes.

The ADAIR process is useful in evaluating the completeness of design support provided by a given approach or system. In fact, the majority of this chapter uses the ADAIR process to structure its evaluations and comparisons. Not all of the reviewed approaches and systems support all phases, but in cases where a given approach or tool does not support a given phase, it can usually be improved by adding support for that phase.

Activate. In the first phase, an appropriate subset of all available critics is selected for activation. Critics that are relevant and timely to the designer's current decisions should be activated so as to support those decisions. Increasing support for activation tends to make the advice provided by the system more useful to designers and reduces the amount of feedback presented that is not useful.

Detect. Second, active critics detect assistance opportunities and generate advice. The most common type of assistance opportunity is the identification of a syntactic or simple semantic error. Other opportunities for assistance include identifying incompleteness in the design, identifying violations of style guidelines, delivery of expert advice relevant to design decisions, or "advertisements" for applicable automation.

Advise. Third, design feedback items are presented to advise the designer of the problem and possible improvements. This phase is central to the concept of supporting

the designer's decision-making. Feedback may take the form of message displayed in a dialog box or feedback pane, or it may take the form of a visual indication in the design document itself (e.g., a wavy, red underline). Much of the potential benefit of critiquing is associated with this phase: the feedback item improves the designer's understanding of the status of the design, the explanation provided improves the designer's knowledge of the domain, and the designer is directed to fix problems. This ultimately results in more knowledgeable designers and better designs. Realizing these benefits requires effective means for designers to manage feedback and careful phrasing of problem descriptions and suggestions.

Improve. Fourth, if the designer agrees that a change is prudent, he or she makes changes to improve the design and resolve identified problems. Fixing the identified error is likely to be one of the most frequent forms of improvement. Other types of improvement clarify the fact that the feedback is irrelevant rather than directly change the offending design elements. For example, the designer might change the goals of the design in reaction to an improved understanding of the problem or solution domain. Design support systems can aid designers in making improvements by providing suggestions for improvements or corrective automations that fix the identified problem semi-automatically.

Record. In the final phase, the resolution of each feedback item is recorded so that it may inform future decision-making. Having a record of problem resolutions is important later in design because each design decision interacts with others. Critics help elicit design rationale as part of the normal design process by acting as foils[1] that give

designers a reason to explain their decisions. A recent evaluation of a critiquing system found that experienced designers often explained their decisions in response to criticism with which they disagreed (Sumner, Bonnardel, and Kallak, 1997).

3.1.4 Comparison of Critiquing Systems

This subsection briefly reviews nine different critiquing systems. Table 3-2 characterizes these critiquing systems according to their support for the phases of the ADAIR process. Each system is given a score from zero to three points for four of the five ADAIR process phases. For the detection phase, each system is described as using comparative, analytic critiquing, or both.

Table 3-2: Summary comparison of critiquing systems

System	ADAIR Critiquing Process Phase				
	Activate	Detect	Advise	Improve	Record
ONCOCIN		Comparative	H		
ATTENDING family		Both	HH	H	
Janus family	HH	Analytic	HH	H	H
Framer	HH	Analytic	HH	HH	
CLEER		Analytic	H		
VDDE	H	Analytic	H		H
TraumaTIQ	HH	Comparative	HH		
AIDA	H	Both	H		
SEDAR	HHH	Analytic	HH	HH	

Comparative critiquing supports designers by pointing out differences between the proposed design and a design generated by alternative means, for example, a planning system with extensive domain knowledge. In contrast, analytic critiquing uses rules to detect assistance opportunities, such as problems in the design.

1. In acting terminology, a foil is a minor character that allows a major character to be expressed through dialog.

Fischer offers the following critic classification dimensions: active vs. passive, reactive vs. proactive, positive vs. negative, global vs. local (Fischer, 1989). Active critics continuously critique the design, whereas passive critics do nothing until the designer requests a critique. Reactive critics critique the work that the designer has done, whereas proactive critics try to limit or guide the designer before he or she makes a specific design decision. Positive and negative critics supply praise and criticism, respectively. Critics that analyze individual design elements are termed local critics, while critics that consider interactions between most or all of the elements in a design are termed global critics. The systems reviewed are split roughly evenly between use of active and passive critics. Only SEDAR provides proactive critics, all other reviewed critiquing systems are reactive. ATTENDING, Framer, Janus, and CLEER offer praise, although it plays a minor role in these systems. On the scale from local to global, a vast majority of the critics in the systems reviewed are near the local end and consider one or a few design elements at a time.

Below, each of these critiquing systems is discussed in roughly chronological order.

ONCOCIN. In 1980, Teach and Shortliffe conducted a survey of doctors' attitudes regarding computer based clinical consultation systems (Teach and Shortliffe, 1981). Some of their conclusions at that time were that (1) doctors are accepting of systems that enhance their patient management capabilities, (2) they tend to oppose applications that they feel infringe on their management roles, (3) such systems need human-like interactive capabilities, and (4) 100% accuracy in the system's advice is neither achievable nor expected.

These findings suggested a new direction for computing systems that support clinical practice (Fagen, Shortliffe, and Buchanan, 1980). These systems follow the traditional expert system user interface paradigm and were evaluated primarily in terms of their knowledge content, rather than their impact on practice. The critiquing concept arose from the realizations that the system should support doctors without infringing on their decision-making authority and that systems that were not 100% accurate could play a useful supporting role.

The next year, Langlotz and Shortliffe reported on the conversion of ONCOCIN, an expert system for the management of cancer patients, to the critiquing approach. Initial versions of the system functioned as an expert system that produced plans that essentially consisted of a set of drugs and dosages. The intended users felt “annoyed” at having to override the system’s advice when they did not agree with the generated treatment plan (Langlotz and Shortliffe, 1983). ONCOCIN was converted into an embedded critic: rather than use the system primarily to generate treatment plans, doctors were intended to routinely enter their own plans into ONCOCIN and the system offered criticism as a side benefit.

The ATTENDING family. At about the same time that ONCOCIN was being developed at Stanford, Miller was developing the ATTENDING system at Yale. Like ONCOCIN, much of the emphasis of ATTENDING was on prevention of the negative effects of the traditional expert system user interface. “ATTENDING avoids the social, medical, and medicolegal problems implicit in systems which simulate a physician’s

thought processes, and thereby attempt to tell him how to practice medicine" (Miller, 1983).

ATTENDING advises an anesthetist in the proper design of an anesthetic plan to be executed during surgery. ATTENDING prompts the designer (in this case, an anesthetist) to enter a description of the problem (a patient's conditions) and a proposed solution. ATTENDING then produces two or three paragraphs of natural language criticism and praise of the plan. Any part of the proposed treatment plan that does not trigger criticism is praised; this is done on the assumption that a more positive tone will enhance acceptance of the tool.

The Janus family. The Janus family consists of several versions of a household kitchen design environment, named successively Crack (Fischer and Morch, 1988), Janus (Fischer et al., 1992), Hydra (Fischer et al., 1993), and KID (Fischer, Nakakoji, and Ostwald, 1995). Designers use these systems by choosing a floor plan layout and placing cabinets, counters, and appliances in that floor plan. One panel of the Janus user interface window shows the current state of the kitchen, while other panels show a palette of available design materials, example floor plans, and feedback from critics. Additional windows are used for argumentation and specification of design goals. A library of IBIS-like arguments about alternative design decisions is available (Fischer et al., 1991). Goal specification sheets prompt the designer to provide information through a structured set of choices, for example, "How large is the family using this kitchen?", and "Is the cook right- or left-handed?" Furthermore, designers using Hydra can select a critiquing

perspective (i.e., critiquing mode) to activate critics relevant to a given set of design issues and deactivate others.

Framer. The Framer design environment (Lemke and Fischer, 1990) supports user interface window layout created with CLIM (the Common Lisp Interface Manager). One panel of the Framer window is used to edit the current state of the design. A checklist panel shows a static list of tasks to be performed in the design process, with one checklist item marked as the current task. A panel titled “Things to take care of” presents the system’s advice for improving the design. Beside each piece of advice are buttons to explain the problem, dismiss the criticism, and, in some cases, automatically fix the problem. The two main contributions of Framer are its use of a process model to activate critics and the fact that it offers corrective automations.

CLEER. Configuration assessment Logics for Electromagnetic Effects Reduction (CLEER) is loosely integrated with a computer aided design (CAD) system for placement of antennas on military ships (Silverman and Mezher, 1992). The placement of antennas on ships affects the performance of the antennas, the radar profile of the ship, and the function of other shipboard equipment. Designers using CLEER position antennas in a CAD model of a ship. When the designer presses an “Evaluate” button, feedback from critics is displayed in a scrolling log window.

CLEER does not automatically activate critics and has no user or design task model. Analytic critics in CLEER detect problems with mechanical and electromagnetic features of the design. Silverman and Mezher propose an enhanced version of CLEER that would

use decision networks to add support for activation, advisement, and improvement (Silverman and Mezher, 1992).

VDDE. The Voice Dialog Design Environment (VDDE) (Bonnardel and Sumner, 1996) is a design environment for voice dialog systems, for example, the menu structure of a voice mail system. VDDE applies stylistic guidelines to help the designer comply with standards, and it can compare two voice dialog designs for consistency with each other.

Critics in VDDE display their feedback as one-line messages in a scrolling log window. A separate control panel window is used to configure the critiquing system. VDDE does not automatically activate critics based on a user or goal model. Instead, designers directly specify which sets of critics should be active, their priorities, and how actively they should be applied. Unlike Hydra, multiple sets of critics can be active simultaneously.

Sumner, Bonnardel, and Kallak (1997) did an exploratory study of four professional voice dialog designers using VDDE. One unexpected observation was that designers anticipate critics and change their behavior to avoid them. This is positive if designers are avoiding decisions that are known to be poor. However, the designer's understanding of the rule may be inaccurate and lead to "superstitious" avoidance of some decisions. The fact that designers rapidly internalize criticism emphasizes the need for each criticism to provide a clear explanation. Another observation was that experienced designers tended not to change their designs in response to criticism. Instead, they stated why they thought that their decisions were correct. This can be interpreted as a negative result in that

suggested changes were not carried out. However, if critics act as foils that prompt designers to externalize their design rationale and expertise, the effect could be exploited to support the recording of design decisions.

TraumaTIQ. TraumaTIQ is a stand-alone system that critiques plans for treatment of medical trauma cases, such as gunshot wounds (Gertner and Webber, 1998). One emphasis of TraumaTIQ is the time-critical nature of its domain.

A doctor or scribe nurse enters treatment orders into the system as they are performed. TraumaTIQ infers the doctor's treatment goals from these orders and generates its own treatment plan. If substantial differences are detected between the generated plan and the entered orders, TraumaTIQ presents a dialog box with a few concise, natural language critiques. Each piece of advice contains a brief explanation and is sorted by urgency in the output window.

AIDA. The Antibody IDentification Assistant (AIDA) is a tool intended for use by medical laboratory technicians to categorize blood samples (Guerlain et al., 1995). The antibody identification task is primarily a problem solving task: the technician must interpret a panel of tests carried out on a batch of blood samples and classify each clinically significant antibody as ruled out, unlikely, likely, or confirmed. In forming a complete solution, technicians must first make a partial solution, use their limited knowledge to evaluate it in terms of how well it explains the data, and then revise their solution.

Traditionally, the identification task is done by filling in a grid on a paper form; AIDA's user interface is centered on an electronic version of this form. A separate critiquing feedback dialog box is presented when the practitioner reaches certain steps in the design process and the proposed solution differs from one generated automatically by the system.

Since AIDA is capable of generating its own solution to most antibody identification problems, one might wonder why a human user is involved in problem solving at all. The reason stems from the fact that the system is not completely competent in solving all problems. If the system were to be totally automated, the human user would still have to solve the problem independently to decide whether to accept the machine generated solution. Humans do a very poor job at this task, and frequently err by assuming that an incorrect solution is correct, or by following the system's explanation "down the garden path" to the same incorrect solution. Furthermore, users of automated expert systems can be expected to reduce their skill level over time due to the lack of practice. However, verifying the correctness of an automatically generated solution to the antibody identification task can require more skill than designing a new solution. Roth, Malin, and Schreckenghost refer to this as the "irony of automation" (Roth, Malin, and Schreckenghost, 1997).

Guerlain et al. evaluated AIDA by asking thirty-two professional laboratory technicians from seven different hospitals to solve four difficult problems (Guerlain et al., 1995). Half of the subjects were assigned to use AIDA with the critics turned on and half worked with the critics turned off. In total, the group that did not use critics had twenty-

nine errors in their solutions, while the group using critics had only three errors. These three errors arose in one of the problems where the system's knowledge was incomplete and it could not generate a correct solution. Despite this incompleteness, the critic-using group still did better on that problem than did the control group, which produced eight errors.

SEDAR. The Support Environment for Design and Review (SEDAR) is a critiquing system for civil engineering (Fu, Hayes, and East, 1997). Specifically, it supports the design of flat and low-slope roofs. Many guidelines for roof design are available to practitioners, yet approximately 5% of roofs constructed in the U.S. fail prematurely, in part because of design errors.

SEDAR is tightly integrated into a CAD program. While designers work with the CAD program to enter their design decisions, critics check the design for problems. The presence of problems is indicated by a status message, and a dialog box that lists outstanding problems can be accessed through a menu. SEDAR can visually suggest a design improvement by drawing a new design element in one corner of the screen with an arrow to the general area where the new element should be placed. However, the designer must still use the normal CAD tool commands to make a new instance of the suggested design element and place it into the design.

SEDAR provides three activation strategies: error prevention, error detection, and design review. The error prevention strategy works before designers commit to certain design decisions, for example, as soon as the designer begins placing a mechanical unit in the design, illegal areas are visually marked-off on the design diagram. The error

detection strategy implicitly applies active critics to the design as changes are made. The design review strategy provides a batch of criticism for use by reviewers after the design is considered complete.

SEDAR is unique among the critiquing systems reviewed here in that it identifies two classes of project stakeholders: designers and reviewers. SEDAR's authors outline a broader design process in which the design document is repeatedly passed between designers and reviewers, causing many project delays. Unfortunately, SEDAR supports each group of stakeholders independently: there are no critics that advise designers how to make designs that are easier to review. For example, there is no critic that warns the designer to avoid using mechanical equipment that is not familiar to the reviewers.

3.1.5 State of the Art of Critiquing Systems

Research on critiquing systems has been motivated by three main observations: (1) in certain domains it is impractical to build expert systems that are acceptable to users, (2) human designers sometimes make costly errors that could be avoided with better tool support, and (3) design is a cognitively challenging task that could be eased with tool support to help designers overcome specific difficulties. The earliest system reviewed, ONCOCIN, was built as a reaction to user rejection of expert systems in the medical treatment planning domain. Most of the reviewed critiquing systems, including CLEER and SEDAR, focus on identifying specific types of errors and trying to warn designers about these errors. The Janus family and the Argo family of design environments address the much broader scope of cognitive support.

The critiquing systems reviewed have primarily been research systems that have seen little practical use. Each system explores some aspects of design support while ignoring others. Also, the critiquing systems reviewed have all been fairly limited in the number of critics and the scope of their domain. To date, no “industrial strength” critiquing system has been implemented and deployed. In part, this is because little work has been done on the software engineering issues of developing reusable infrastructures, development methodologies, or authoring tools for creating critiquing systems. Argo/UML is the first design critiquing system to successfully scale up in terms of complexity and in the size of its user base.

Overall, existing critiquing systems provide incomplete support for designers’ cognitive needs. In most of the systems reviewed, design critics detect and highlight errors, but they require designers to do much of the work of activation, feedback management, design improvement, and recording.

CHAPTER 4: Proposed Cognitive Features

As described in Chapter 1, my basic research method has been to (1) find and understand published cognitive theories of design, (2) invent new design tool features that address the cognitive needs identified by the theories, (3) build design tools that include these features, and (4) evaluate the impact of these features on designers. I have tried to build design tools that are realistic test-beds for the cognitive support features, i.e., the latter two tools described below are full-scale, useful tools, not toy examples or prototypes.

To date, I have constructed four design tools with cognitive support. Argo/C2 (Figure 4-1) is a software architecture design environment. Argo/C2 includes critics that

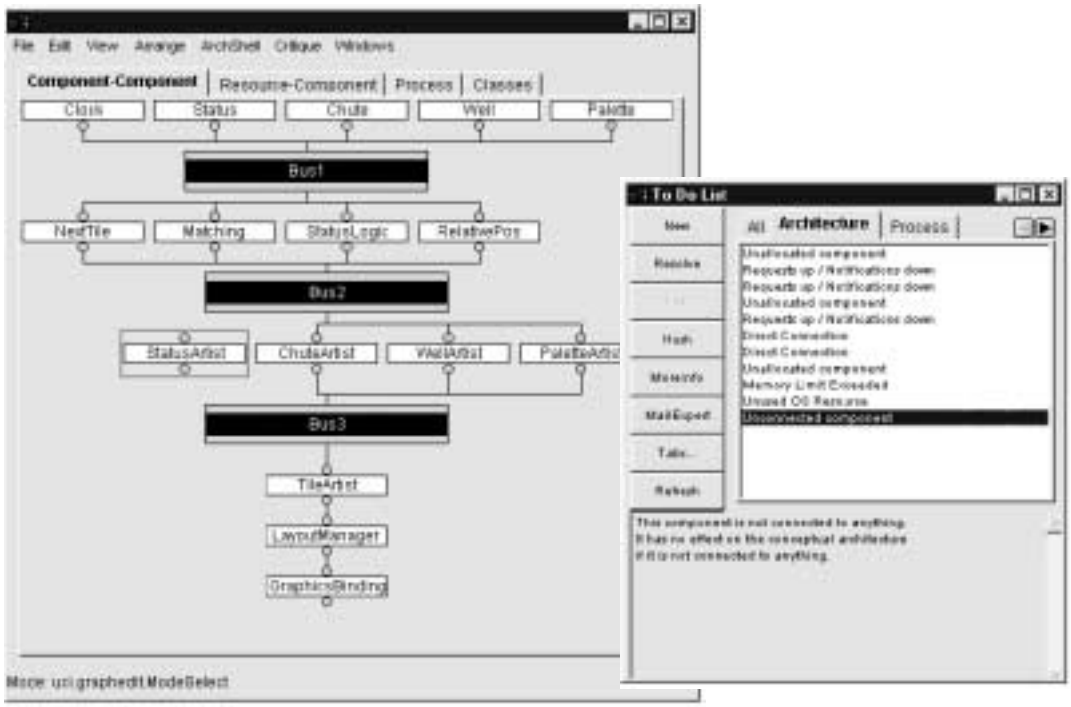

Figure 4-1. Argo/C2: a design tool for C2-style architectures

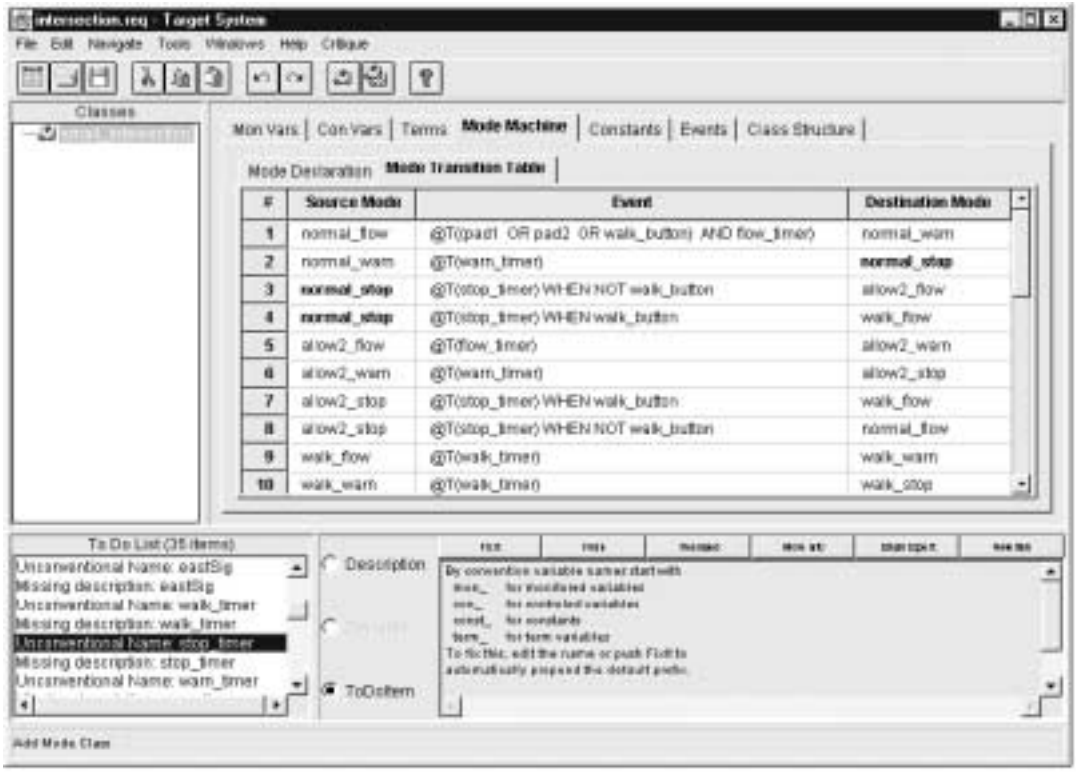

Figure 4-2. Prefer: a requirements tool using the CoRE notation

remind software architects of the C2-style guidelines (Taylor et. al, 1996), a dynamic “to do” list, and a process model. Stargo is an object-oriented design tool using the OMT (Object Modeling Technique) notation (Rumbaugh et al., 1991). Stargo includes critics and a dynamic “to do” list. Prefer (Figure 4-2) is a state-based requirements specification tool using the CoRE notation (Faulk et al., 1994), which is based on the SCR notation (Henninger, 1980). Prefer includes design critics and a dynamic “to do” list.

Argo/UML (Figure 4-3) is the fourth and most ambitious tool. It is an object-oriented design tool using the UML (Unified Modeling Language) notation (OMG, 1997). Argo/UML’s user interface consists of four panes named (clockwise from upper-right) the main pane, the details pane, the “to do” pane, and the navigator pane. The main pane is used for drawing design diagrams and editing tabular views of the design. The details pane

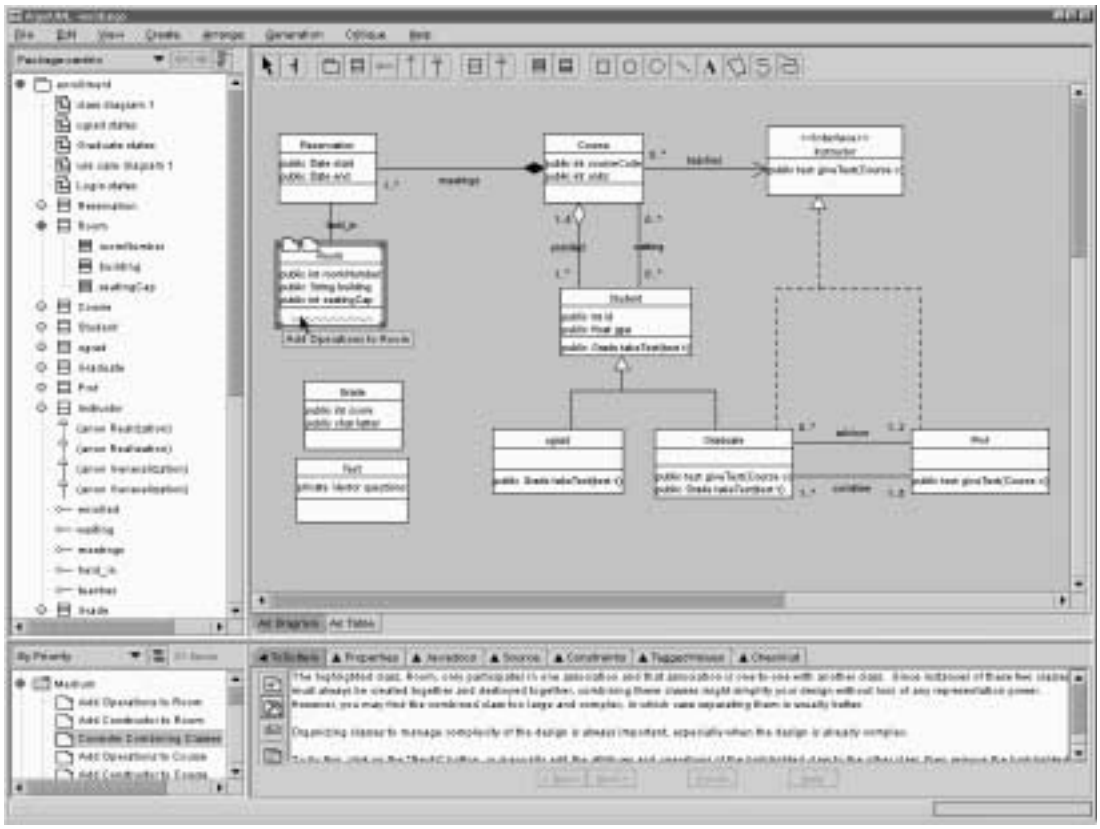

Figure 4-3. Argo/UML: an OODA tool using the UML notation

contains several tabs. Most of the “details tabs” have an upward pointing arrowhead and show information about the currently selected object in the main pane. However, the first details tab has a left facing arrowhead and shows information about the currently selected object in the “to do” pane. The “to do” pane contains the designer’s “to do” list (described below). The navigator pane allows the designer to see all the elements of the design and their relationship to other elements in the context of the selected navigational perspective (described below).

Argo/UML is a useful tool and a showcase for over a dozen cognitive support features. These features are covered in the following subsections and summarized in Table 4-1. For each feature, I provide background information, describe how the feature is

used, and explain its theoretical and experiential motivations. This chapter concludes with a discussion of how the features interact.

Table 4-1: Summary of proposed cognitive features

Knowledge Support Features

§4.1.1.	Critics and Control Mechanisms	Implemented, Deployed, Evaluated
Catch design errors early and provide constructive advice		
§4.1.2.	Non-modal Wizards	Implemented, Deployed, Evaluated
Provide procedural guidance to resolve identified problems		
§4.1.3.	Context Sensitive Checklists	Implemented, Deployed
Help catch design errors early, less specific than critics		
§4.1.4.	Design History	Partly Implemented
Helps review past criticisms, manipulations, and resolutions		

Process Support Features

§4.2.1.	"To Do" List and Clarifiers	Implemented, Deployed, Evaluated
Presents criticism and advice in a usable, organized format		
§4.2.2.	Opportunistic Search Utility	Partly Implemented, Deployed
Helps designers find requested design elements and additional related elements		
§4.2.3.	Opportunistic Table Views	Implemented, Deployed
Dense, task-specific views that facilitate systematic scanning and data entry		

Visualization Support Features

§4.3.1.	Navigational Perspectives	Implemented, Deployed
Tree-structured views of the design emphasizing alternative relationships		
§4.3.2.	Broom Alignment Tool	Implemented, Deployed, Evaluated
Helps designers establish and maintain alignment as secondary notation		
§4.3.3.	Model-based Layout	Described, Mock-up
Automatic diagram layout that emphasizes semantic properties of the design elements		

Construction Support Features

§4.4.1.	Selection-Action Buttons	Implemented, Deployed, Evaluated
Context-sensitive buttons that provide easy access to common construction actions		
§4.4.2.	Create Multiple	Described, Mock-up, Evaluated
Rapidly create design elements by instantiating reusable design fragments		
§4.4.3.	Visual Blender	Described, Mock up
Inspires creative design decisions by visually combining design concepts		

4.1 Knowledge Support Features

4.1.1 Design Critics and Criticism Control Mechanisms

Background. Critics are active agents that continually check the design for errors or areas needing improvement. Critics can deliver knowledge to designers about the implications of, or alternatives to, a design decision. Critics simply advise the designer; they do not prevent the designer from taking action. In this way, they support the designer in working through invalid intermediate states of the design. 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. Feedback is often valuable when it addresses issues that the designer had previously overlooked and might never seek to investigate without prompting.

Each critic performs its analysis independently of others. Each checks one predicate and delivers one piece of design feedback. Critics encapsulate domain knowledge of a variety of types. Correctness critics detect syntactic and semantic flaws. Completeness critics remind the designer of incomplete design tasks. Consistency critics point out contradictions within the design. Optimization critics suggest better values for design parameters. Alternative critics present the designer 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 designer 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 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. Table 4-2 shows some of the object-oriented software design critics implemented in Argo/UML.

Table 4-2: Examples of critics in Argo/UML

Priority	Knowledge Type	Headline
High	Semantics	Remove {name}'s Circular Inheritance
Medium	Alternative	Consider Combining Classes
Medium	Completeness	Add Operations to {name}
Medium	Completeness	Add Trigger or Guard to Transition
Medium	Completeness	Choose a Name
Medium	Completeness	Define Class to Implement Interface {name}
Medium	Evolvability	Reduce States in Machine {name}
Medium	Presentation	Make Edge More Visible
Medium	Presentation	Revise Name to Avoid Confusion
Medium	Semantics	Remove Aggregate Role in N-way Association
Medium	Semantics	Remove Unneeded Realizes from {name}
Medium	Syntax	Capitalize Class Name {name}
Medium	Syntax	Revise Package Name {name}
Medium	Tool	Change Multiple Inheritance to Interfaces
Low	Alternative	Consider using Singleton Pattern
Low	Consistency	Singleton Stereotype Violated

Formalizing the analyses and rules of thumb used by practicing software designers could produce hundreds of critics. To provide the designer with a usable amount of information, a subset of these 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 designer. Specifically, designers should be able to easily view relevant and timely feedback items without having to sort through

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, and ideally, should be short enough to maintain a feeling of inter-activity.

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 design decision it supports. Criticism control mechanisms check those attributes against the design goals and process model. Computing relevance and timeliness separately from critic predicates allows critics to focus entirely on identifying problematic conditions in the product (i.e., the partial design) 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.

Description. Designers using Argo/UML do not usually interact with critics or criticism control mechanisms directly. Instead, they simply see the feedback produced by critics presented via the “to do” list and clarifiers (described below). However, designers can directly edit the user model that is used by criticism control mechanisms. Figure 4-4 shows Argo/UML’s decision model editor: designers can prioritize the types of decisions involved in object-oriented design, and the critics that support those decision types will be activated or deactivated based on that model. Also, designers can enable or disable individual critics by using the Critic Browser window (Figure 4-5).

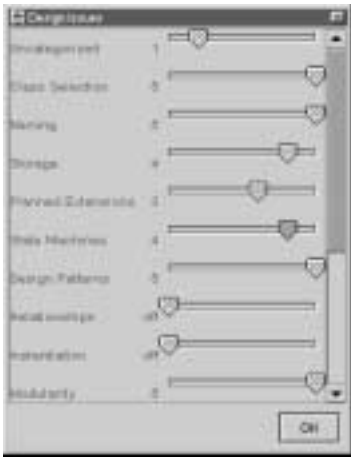

Figure 4-4. Decision model editor

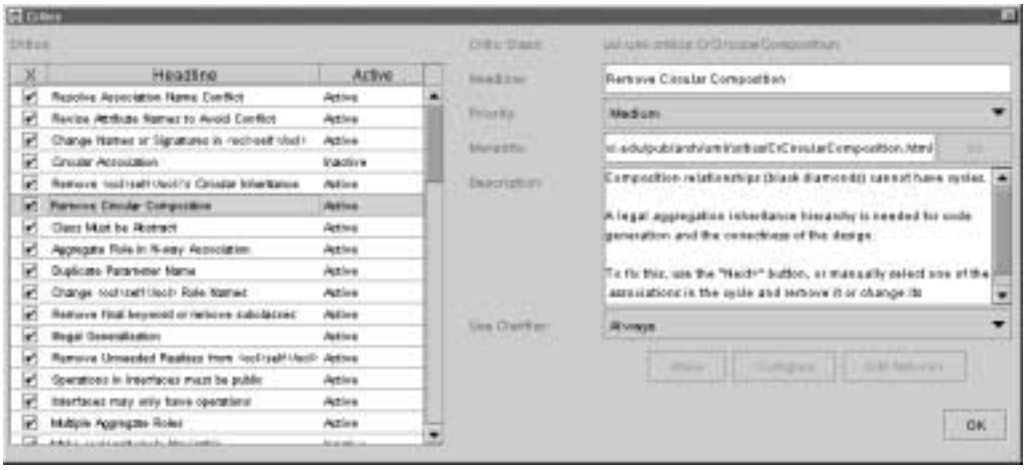

Figure 4-5. Critic browser window

Mapping to theory. Critics are motivated by the theories of reflection-in-action and opportunistic design. Critics provide automated support for reflection-in-action by doing some of the analysis work that would otherwise be the responsibility of the designer. This can help average designers work like expert designers because the critics prompt them to consider the same issues that expert designers consider. Critics can also help all designers avoid slips or oversights than can occur when working under pressure.

The theory of opportunistic design predicts that when a designer is blocked by not knowing how to solve a problem they encounter, they tend to switch to an alternative design task. Critics can help designers avoid context switches and follow through on their original design plans if the critic detects the problem and offers the designer advice leading to a solution. However, designers will naturally switch tasks opportunistically during the course of design. When they do, critics can provide a "safety net" that allows them to deviate from the specified process without fear of forgetting to complete all details and correct all problems as they go.

Furthermore, the automatic application of critics addresses the confirmation biases and fixation effects that designers may experience during construction of designs. Alternative critics prompt designers to consider specific alternatives to decisions they have made. Ideally, the suggested alternative is itself an improvement. However, considering the alternative can cause the designer to activate previously inactive memory structures and bring new parts of his or her knowledge into play. This can help designers break out of the cyclic memory activations associated with fixation, even if the critic does not directly offer a better solution.

Possible extensions. No widely useful critic implementation language has been proposed to date. A potential extension to this dissertation would explore the possible advantages of special purpose critic languages as compared to the use of general-purpose languages like Java. Many researchers have investigated end-user programming (e.g., Girgensohn, 1992; Riesbeck and Dobson, 1998). However, no critic language has been successfully demonstrated as useful to practicing designers. One key goal and point of

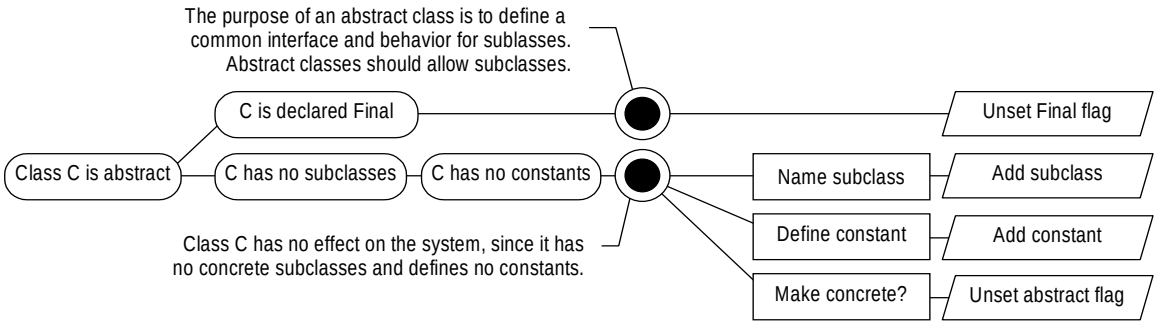

Figure 4-6. Proposed graphical specification of critics and wizards

comparison is the capability for practicing designers to easily specify improvements to critics. Figure 4-6 shows my proposed graphical notation for specifying critics and wizards. The flow of control starts at the left-most node and proceeds to the right, taking all branches, until a condition is not satisfied. Rounded rectangles indicate conditions that must be satisfied. If control reaches a bull's-eye node, the critic fires and generates feedback. Rectangular nodes describe user interface panels presented to the user as a step in the wizard. Parallelogram nodes are actions that modify the design or user model. It is expected that practicing designers using this notation will be able to easily propose changes that add a new case where the critic should fire or that place a new restriction on an existing case.

4.1.2 Non-modal Wizards

Background. 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 in UML; one simple and automatable solution to this problem is to capitalize the first letter of the class name. Not

all solutions can be implemented in a single step, however. Some solutions 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 suggested fix involves several steps to convert one superclass into an interface and move method definitions down into subclasses. Argo/UML's non-modal wizards aid designers in solving identified problems by guiding them through the steps of the solution without unnecessarily constraining them.

Description. 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 development tools and desktop applications: they guide the designer through a sequence of steps and decisions in a predefined task (Dryer, 1997). A wizard typically performs design manipulations on the designer's behalf; but in some cases, suggested fixes consist solely of step-by-step instructions (i.e., cue cards) 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 PostIt note icon for the affected feedback item in the "to do" list.

Unlike wizards found in other tools, Argo/UML's wizards are non-modal and apply changes immediately rather than at the final step. The designer is free to leave the wizard at any time to perform direct manipulations on 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 necessary for wizards that simply direct the designer through a

series of manual steps. Non-modal wizards also allow designers to opportunistically perform other design manipulations that are logically related to the steps of the wizard. For example, in working through a wizard for removing multiple inheritance, the designer may choose to work outside of the wizard to move some methods to an entirely different location in the inheritance hierarchy. Once the designer has begun using a wizard, the “to do” item that gave rise to the wizard will not be removed until the wizard is finished or canceled.

Non-modal wizards in Argo/UML provide a spectrum of investment choices ranging from low effort cue cards to traditional wizards with substantial automation to push-button corrections. As with checklist and critics, the tool builder and the organization using the tool can make the decision to invest more effort in knowledge support based on feedback from designers using the initial low investment versions. Those cue card wizards that are found useful then become candidates for further investment in automation. If a given wizard proves very often useful and requires no additional information from the designer, it may even be applied automatically to correct problems without explicit confirmation.

Argo/UML’s support for non-modal wizards is also motivated by a concern for the authoring cost needed to build them. Argo/UML provides several options for wizard authoring with a range of cost and value. Simple suggestions on how to solve the identified problem can be authored as textual cue cards without programming. From there, partial automation and some wizard user interface elements can be provided by reusing existing wizard steps. Custom user interface elements or new automated design

manipulations can be built by extending the existing framework. Finally, “push-button” automated solutions may be developed for some kinds of design problems. The decision to invest more effort in additional automation can be made based on experience with earlier versions.

Mapping to theory. Wizards in Argo/UML and other tools address the fact that designers have limited knowledge. While the problem descriptions generated by critics augment designers’ analytical knowledge, wizards augment designers’ procedural knowledge of how to fix problems. This procedural knowledge exists at two levels: general design manipulation strategies and specific tool commands. Each of these levels of procedural knowledge is discussed below. The cognitive theories of reflection-in-action, opportunistic design, and comprehension and problem solving also helped inspire this feature.

At the strategic level, experienced designers are likely to posses a well-stocked library of design manipulation strategies. For example, when replacing a design element with a new one, it is usually best to configure the new element while referring to the existing one, then delete the old element rather than deleting the old element first and relying on one’s short-term memory to construct a corresponding new element. Wizards can contain knowledge about effective strategies and guide designers in following them. This supports less experienced designers who may not possess appropriate strategies, and it helps all designers execute strategies under stress or when distracted. As discussed in Section 2.2.4, Guindon, Krasner, and Curtis (1987) identified the lack of knowledge about design strategies as one of the main difficulties facing large software design projects.

At the tool-specific level, even experienced designers may lack knowledge of new or rarely used design tool features. For example, if a wizard tells the designer to access the pop-up menu on a certain design element, the designer will learn that a pop-up menu is available; this might not have been obvious otherwise. In addition to teaching designers about the design tool's user interface on demand, wizards may also provide special-purpose user interfaces that are not otherwise available. For example, if one step in a wizard requires the designer to move methods from one class to another, it can present a user interface with two scrolling lists of methods and method movement buttons. Keeping such rarely used, special-purpose user interface panels in wizards and out of the main menus and toolbars helps to reduce the apparent complexity of the tool and may lower initial learning costs. The usefulness of special-purpose, task-based user interfaces is also indicated by the theory of comprehension and problem solving. In particular, a wizard step can aid designers' comprehension of their design if it brings together design elements to highlight an interaction or design trade-off that is not clear in task-independent views.

Based on the theory of reflection-in-action, I chose to make Argo/UML's wizards take action in each step so that the designer can reflect on the implications of each design decision as it is being made. Since design decisions interact, it is likely that a change to one part of the design will force the designer to consider a cross-cutting issue that affects other parts as well. Once the designer is considering the cross-cutting issue, it may be easier for him or her to make a design excursion to deal with all the affected design elements before continuing on with the initial wizard. Based on the theory of opportunistic design I suspected that normal, modal wizards might force the designer to follow through on a potentially costly train of thought. By making wizards non-modal,

Argo/UML allows designers to switch tasks to pursue those with lower cognitive costs. Non-modal wizards and the “to do” list help designers return from design excursions by keeping partially resolved items in the “to do” list and indicating them with a visible progress bar.

4.1.3 Context Sensitive Checklists

Background. Conducting design reviews and inspections is one of the most effective ways of detecting errors during software development. In a recent editorial, Glass (1999) reviewed the results of controlled studies on the effectiveness of inspections and summed up the three best software engineering practices as “inspections, inspections, inspections.” A design review typically consists of a small number of designers, implementers, or other project stakeholders holding a meeting to review a software development artifact. Many development organizations have developed checklists of common design problems for use in design review meetings. Porter and Johnson (1997) found that reviewers inspecting code without meeting were just as effective as design review meetings. I have added a checklist feature to Argo/UML that is much in the spirit of design review checklists. However, Argo/UML’s checklists are integrated into the design tool user interface and the design task.

Description. A software designer using Argo/UML can see a review checklist for any design element. The “Checklist” tab presents a list of check-off items that is appropriate to the currently selected design element. For example, when a class is selected in a design diagram, the checklist tab shows items that prompt critical thinking about classes (Figure 4-7). Designers may check off items as they consider them. Checked items are

Figure 4-7. Context sensitive checklist

kept in the list to show what has already been considered, while unchecked items prompt the designer to consider new design issues. Argo/UML supplies eleven different checklists with two hundred possible items in total.

The items in the list are phrased in concrete and specific terms whenever possible. For example, “Does the name ‘Student’ clearly describe the class?” and “Is ‘Student’ a noun or noun phrase?” In contrast, paper-based checklists must use generic terms (e.g., “Is the name of the class a noun or noun phrase?”) and rely on human interpretation of those terms. Clearly humans are able to interpret generic terms; however, doing so is an additional cognitive operation that can be avoided with tool support.

Each checklist item can have a guard condition that determines if the item is appropriate based on the design context. A checklist item is only shown to the designer if its guard expression evaluates to true; this helps keep checklists of manageable size and increases their perceived relevance. For example, one checklist item for Attributes (i.e., instance variables of a class) prompts the designer to consider breaking complex variables down into parts (e.g., a telephone number could be represented as a single attribute, or it could be broken down into area code, prefix, number, and extension). The guard condition for this item checks that the selected attribute’s type is not boolean, since

booleans cannot be broken down into smaller parts. Unlike a critic's predicate, the guard condition of a checklist item is optional. Checklists without guard conditions are used to prompt the designer to consider issues that are frequently useful but that cannot be evaluated by the system.

Mapping to theory. Checklist items are motivated by many of the same cognitive needs that motivate critics and the dynamic "to do" list (described below). Like critics, checklists support reflection-in-action by asking questions about the design that the designer might not ask on his or her own. Like "to do" lists, checklists support opportunistic design by listing out issues and allowing the designer to choose which issue to address next.

Checklist items tend to be generic prompts for the designer to apply their own knowledge. In contrast, critics typically identify more specific problems and supply knowledge that the designer may lack. These generic prompts can lead to distraction by raising issues that require knowledge that is very different than what the designer has in mind. This potential disadvantage is mitigated by making checklists less intrusive. The presentation of checklist items (Figure 4-7) is much less intrusive than the clarifiers (described below) and "to do" list items that are used to present feedback from critics.

Checklists are also motivated by the need for a low cost way to build up Argo/UML's knowledge base of advice. None of the critiquing systems discussed in Chapter 3 included large knowledge bases, in part because critiquing systems require the critic author to fully specify each critic at considerable effort. In contrast, Argo/UML supports several authoring options with a range of effort and value. Checklist items that do not have guards

can be authored with the lowest cost since they are simply textual. I expect that in many cases an organization's existing design review checklists can be put into Argo/UML by simple cut and paste. Once a checklist item has been defined, a guard condition can be added with a single line of code. The next step up in terms of effort and value is a critic with an initial predicate. From there the predicate can be refined to increase the critic's relevance. Additional value in the form of detailed explanations, special clarifiers (described below), cue cards, or wizards can be achieved with incremental effort. The decision to invest effort in refinement can be based on experience and feedback from designers.

Possible extensions. Checklists could be made more useful by collecting and summarizing the results of informal design reviews. For example, designers might specify a confidence level rather than a simple checkmark for some checklist items. These confidence levels would then be summarized and ranked so that the parts of the design with the lowest designer confidence could be examined again. Checklists might also be extended by adding support for wizards that help the designer understand the details of the issue raised, and that provide automation to aid the designer in achieving the desired design quality. These checklist wizards would have many of the same advantages as the wizards associated with critics; namely, they would provide many task-specific user interfaces without complicating the normal user interface and would also be non-modal.

4.1.4 Design History

Background. Because design decisions are interrelated, rationale for past decisions is a key part of the design context of new decisions. For example, a software architect

building an HTML editing application might initially choose the most full-featured implementation of a table editing component, only to find that it is incompatible with the spell-checking component. In deciding how to resolve the problem, the architect must know why that particular spell-checking component was used. Blindly replacing the spell-checking component with a more flexible one risks violating the implicit assumptions of related decisions.

Design history is a time-ordered list of the events leading up to the current state of the design, including design manipulations, criticisms offered, and criticisms resolved. I intend a design history to include less information than a design rational in that a history focuses on *what* happened and need not address in detail *why* things happened. Certainly, knowing why past design decisions were made would provide stronger support for current decisions; however, the cost of recording what happened is much lower, and seeing what happened can cue the designer's memory of why certain decisions were made. Furthermore, including criticism and criticism resolutions in the history helps to capture some of the reasons why certain design changes were made.

In addition to informing current decision-making, design history is also needed to avoid repeating any criticism that is resolved by actions outside of the design tool. For example, if the spell-checking component is a "beta" version rather than a fully tested product, then an organizational critic might advise the designer that all use of beta components require special commitments from the quality assurance manager. The designer might discuss it with the manager and agree that it would be acceptable to use the beta version in this case. The designer would then dismiss the critic's "to do" item,

possibly entering a brief explanation. The same criticism should not be presented again for the spell-checking component, despite the fact that the design is in the same state that caused the critic to fire initially.

One challenge faced by design rationale systems is that designers may not take the time to enter information (Lee, 1997). Critics help elicit design rationale as part of the normal design process by acting as foils that give designers a reason to explain their decisions. A recent evaluation of a critiquing system found that experienced designers often explained their decisions in response to criticism with which they disagreed (Sumner, Bonnardel, and Kallak, 1997).

Description. The “History” tab at the bottom of the Argo/UML main window shows a time-ordered list of design history items. Items are recorded for each “to do” item raised, each design manipulation performed, and each “to do” item resolved. These “to do” items normally contain criticisms from design critics, and their resolutions normally contain comments from designers that justify decisions or links to history items for manipulations that resolved the problem.

The history list can display all history items in time order or it can be focused on just those history items that relate to the selected design element. The designer can review design history by selecting an item in the history list on the left side of the “History” tab. Doing so displays a description of the history item in the text area at the right and updates a list of related design elements. Clicking on a related design element will select that element in the diagram pane and allow the user to view its properties.

Figure 4-8. Argo/UML's feedback item dismissal dialog

In addition to building a design history, resolutions to identified problems are sometimes used to adjust Argo/UML's user model and goals model. For example, if the system under design is only intended for experimental use, the criticism that beta components require special testing commitments might be resolved by the architect pressing the "Dismiss" button and choosing "It's not relevant to my goals" (Figure 4-8). In response, Argo/UML immediately opens the goals model window so that it may be updated. If the architect chooses "It's not of concern at the moment," Argo/UML opens the decision model window. Keeping the user model and goals model accurate requires frequent updates. Yet, we cannot assume designers will make the effort to update them without prompting.

Mapping to theory. Argo/UML's design history feature is inspired by the observations that design decisions are interrelated, that designers may have difficulty recalling past design decisions when that knowledge is needed, and that providing the right cues can aid in associative memory retrieval. Furthermore, to the extent that a design history support building a design rationale, the design history can help inform new

decisions with knowledge about past decisions that the designer would not otherwise possess.

Possible extensions. Future extensions to this feature might include a way for the tool to infer causal links between the design manipulations and the problems that they cause or resolve. Also, the design history user interface could provide better views of the design history that better suit specific recurring tasks. The same basic ideas used in the navigational perspectives on the design itself, could be used to construct task-specific views of the design history.

4.2 Process Support Features

4.2.1 Dynamic “To Do” List and Clarifiers

Background. Once a critic generates design feedback, that feedback must be presented to the designer in a usable form without unduly distracting the designer from the task at hand.

Some current CASE tools, such as Rational Rose, follow the familiar paradigm of textual compiler error messages. The difficulty with textual feedback logs is that they are usually structured linearly in the order the messages were generated and cannot be organized according to the designer’s interests. Furthermore, textual error messages are either too short or too long: they cannot be used effectively to both teach concepts to designers who lack needed knowledge and to conveniently prompt designers who have the needed knowledge but were not able to recall it at the time needed. Also, textual error logs are normally presented in a scrolling text widget that is distinct from the design

diagram drawing area. Section 7.1 summarizes the result of a pilot Argo/UML user study which indicated that designers tend not to move their eyes from the design diagram to other panes of the same window.

Description. In Argo/UML, the “to do” list user interface presents feedback to the designer. The “to do” items on the list are grouped into categories, for example, by priority, by design decision type, by offending design element, or by critic knowledge type. The designer can choose the categorization scheme from a menu above the “to do” list. A count of items on the “to do” list is displayed next to this menu. If the number of items is above 50 or 100, then the count is displayed on a yellow or red background to make it more evident. When the designer selects a pending feedback 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. If the designer double-clicks on a “to do” item, Argo/UML jumps to the offending design elements: the diagram with the offenders becomes the currently displayed diagram.

The designer may use the toolbar buttons in the “ToDoItem” tab 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 e-mail to the person who authored the critic, or dismiss the feedback item. Links to background information and e-mail contact with expert designers provide a design context that the designer can use to 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. Clarifiers were added 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. Argo/UML uses wavy, red underlines (a familiar indication of spelling errors) to indicate errors that occur at a specific part of a design element. A yellow PostIt note icon is used to indicate errors that relate to an entire design element. Errors related to missing or invisible design elements currently do not have clarifiers. Visual cutter is limited 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. A feedback item headline is displayed as a tool-tip if the designer briefly positions the mouse pointer over a clarifier.

Mapping to theory. The dynamic “to do” list user interface supports cognitive needs identified by the theories of reflection-in-action and opportunistic design. According to the theory of reflection-in- action, when designers reach a breakdown (a point in the design session where they are not immediately able to move forward), they may instead reflect on the current state of the design. Argo/UML’s dynamic “to do” list helps designers evaluate the current status of the design by listing potential errors.

The presentation of the outstanding item count and its color are indications of how confident the designer should be in his or her design. This makes accessible an important

aspect of the design that is not visible in the design document itself, and cues designers to make meta-cognitive decisions, such as when to switch from construction to reflection.

Argo/UML's dynamic "to do" list also supports opportunistic design. As mentioned in the discussion of critics, the "to do" list aids designers in opportunistically switching between tasks by providing a list of suggested tasks to choose from and by providing a "safety net" that allows designers to more freely switch tasks with less fear of skipping needed steps.

Clarifiers serve the same basic purpose as the dynamic "to do" list and address the same cognitive needs of designers. Clarifiers help address the limited visual scope of the human eye by providing visual feedback in a location where the designer is focusing his or her attention.

Related work. Rogers (1995) describes an automatic "to do" list used in the CORRECT requirements tool that is very similar to the "to do" list used in Argo/C2. The window consists of two panes: an upper pane that lists all pending "to do" items and a lower pane that shows the details of the selected item. Rogers also suggests that the details should include instructions on what the designer should do to fix the problem and that the tool might offer to fix simple problems automatically. The dynamic "to do" list in Argo/C2 has these capabilities. It also allows the designer to send email feedback to experts, add personal reminders, and to dismiss or hush particular items. Argo/UML takes several steps further by providing alternative perspectives on the "to do" list and automatically customizing the text of each item to the particular identified problem.

Clarifiers and non-modal wizards are closely linked to Argo/UML's "to do" list items and provide additional support not found in CORRECT.

4.2.2 Opportunistic Search Utility

Background. Design documents are complex webs of design elements and relationships. These relationships include explicit relationships for the structure and behavior of the system being designed. However, design documents also include implicit constraints and dependencies between design elements that must be maintained. In some cases these implicit relationships can be made explicit in the design document, but that approach can only be carried so far before the document becomes cluttered and difficult to work with.

Many software analysis tools have been built to produce dependency graphs as output. Two difficulties with these tools is that their graphs rapidly become out-of-date as the software is changed, and there is some effort involved in knowing how to use the analysis tool, recognizing that its results are needed, and interpreting the results.

The related element generation rules used in Argo/UML's opportunistic search utility stand in the same relationship with traditional dependency analysis tools as design critics do with traditional design error detection tools. Both cognitive support features address the same basic goals as their traditional counterparts. However, both cognitive support features are much better integrated into the design tool and the design process.

Description. Argo/UML provides a search utility that works in a way familiar to MS Windows (tm) users: the top part of the search window consists of several tabs where

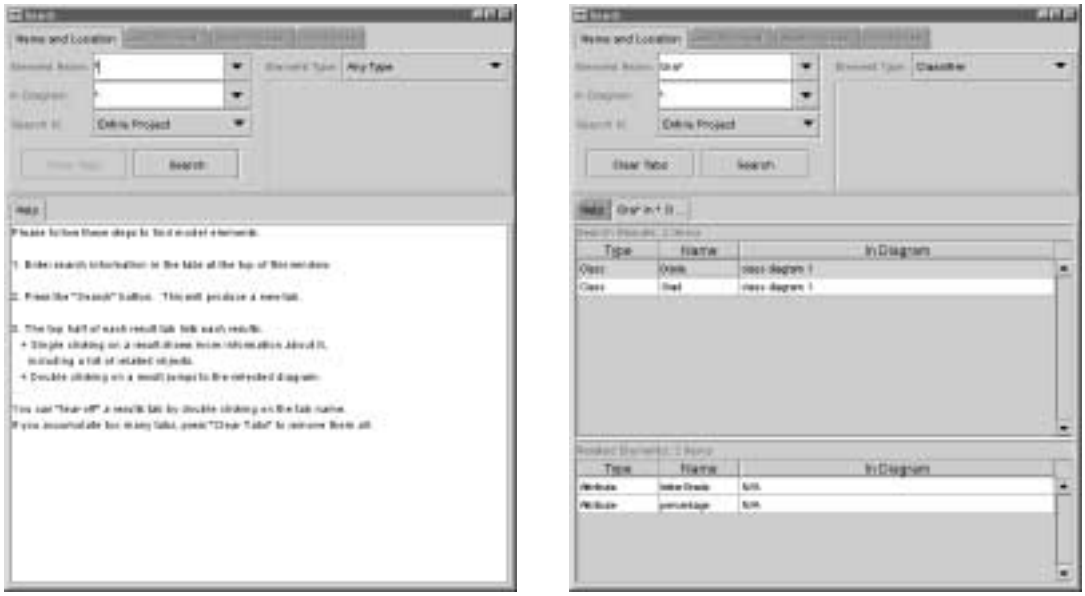

Figure 4-9. Argo/UML's opportunistic search utility window

different search criteria are entered; the bottom part of the search window shows search results. Clicking once on a query result selects it. Double clicking on a query result causes the main Argo/UML window to display a diagram that contains the selected design element.

The main special feature of Argo/UML's opportunistic search utility is the list of elements shown below the query results (Figure 4-9). This list contains design elements that are related to the query result selected in the upper list. The related design elements include those that are likely to need updating if the selected query result is updated, or that should be checked before modifications are made. The set of related design elements are generated using predefined rules provided by a domain expert. These rules could include executing external software analysis tools to produce dependency graphs; however, such rules have not been implemented in Argo/UML.

One other interesting aspect of Argo/UML's search utility is that it stores multiple sets of answers. The bottom half of the window is a tab widget and a new tab is added to store the results of each search. Tabs can also be deleted by pressing the "Clear Tabs" button or "torn off" by double clicking on the tab label.

Mapping to theory. At a practical level, Argo/UML's opportunistic search utility can prompt designers to consider related model elements that they would otherwise be very likely to skip. Like several of Argo/UML's visualization features, the opportunistic search utility makes explicit some design relationships that are needed for specific design tasks but that are not visible in the design notation itself.

On the cognitive level, Argo/UML's opportunistic search utility supports opportunistic design by offering designers alternatives that are likely to be related to their current mental context. The cognitive theory of opportunistic design predicts that designers will often prefer to pursue related design tasks immediately if they require the same mental context, i.e. they have low cognitive cost.

On the other hand, presenting design elements that are weakly related to the current task can distract designers from their current task by prompting them to change their mental context, causing many mental context switches and the associated cognitive cost. To mitigate this potential disadvantage, Argo/UML presents the related elements via a user interface that keeps the original query results visible. This was done to aid the designer in returning from small design excursions.

The opportunistic search utility may also help average designers work more like experts if it prompts them to activate the same memory structures that experts activate. The various features of Argo/UML that help designers think more like experts are discussed more in scene 4 of the usage scenario in Chapter 5.

Possible extensions. An interesting future extension to this feature would be to automatically infer the related element generation rules from logs of expert usage. This would make the feature similar to "recommender systems" found on the internet, such as Yenta (Foner, 1997). Inferring related design elements based on experience might lower authoring costs while increasing the relevance of provided elements. At the least, it would provide a way to check the related element rules offered by experts.

4.2.3 Opportunistic Table Views

Background. Most widely used software design notations, including the Unified Modeling Language, are primarily diagrammatic notations. Diagrams are effective ways of communicating designs because they have immediate visual impact and several secondary notation possibilities. However, diagrams have low visual density, which has been identified as a limiting factor in the adoption of visual programming languages. Furthermore, diagrams tend to be rather difficult to edit and systematically scan. The high effort needed to construct a diagram in most CASE tools encourages designers to use a given diagram for more design tasks than those for which it is well suited. Several of the features described in this dissertation address the weaknesses of diagrams by improving the way diagrams are constructed or by complementing them with non-diagrammatic design views.

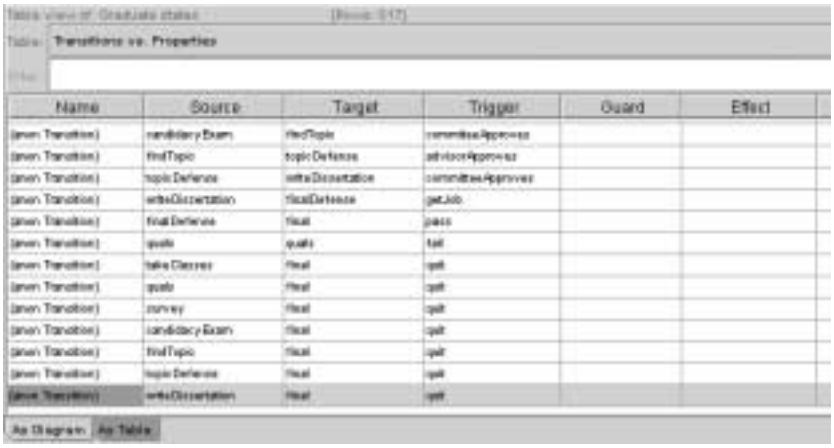

Figure 4-10. Tablular view of state machine transitions

Description. Argo/UML supports UML class diagrams, state diagrams, use case diagrams, activity diagrams, and collaboration diagrams. It complements these diagrammatic 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 of a state machine 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 4-10).

As with familiar spreadsheet interfaces, one cell is considered the active cell at any given time and the cursor keys and mouse can be used to move the current cell. In addition to highlighting the active cell, the entire current row or column may be highlighted to help the designer keep track of the type of systematic scanning he or she is doing. The “Instant Replay” button provides an “instant replay” of recent activity: it briefly highlights the most recently active cells and their entire rows or columns in the same order that the designer accessed them.

Mapping to theory. As with navigational perspectives (described below), one of the goals of Argo/UML's tabular view feature is to provide views that support common design tasks. In particular, tables are easier to systematically scan or fill in than are most diagrams.

However, designers work opportunistically as well as systematically. For example, if a designer is systematically checking that each state has sensible entry and exit actions and finds one with a problematic assumption, he or she may opportunistically switch to looking over the entire design for other elements which depend on that same assumption. In the best case, a design excursion may merely cause the designer to switch from horizontal to vertical scanning in the same table. In the more general case, the excursion may cause the designer to access other diagrams or tables. These design excursions are natural and common; unfortunately, returning from an excursion imposes the cognitive difficulty of recalling one's prior plans. "Instant replays" are one kind of visual prompt that can help the designer recall a previous mental context.

4.3 Visualization Support Features

4.3.1 Navigational Perspectives

Background. The UML standard defines several different diagram types: each of these presents related design elements in an appropriate notation and supports specific design tasks. The set of UML diagram types is based on experience with previous object-oriented design notations and the practical needs of designers. I chose to use the UML standard set of diagram types rather than invent new ones. Designers may spend a significant fraction of time working with one diagram. Nonetheless, different diagrams in

the same design document relate to each other, and when building complex designs, the designer will eventually need to build mental structures that combine elements from multiple diagrams. For example, designers may need to mentally relate elements in two diagrams of different types. Also, related elements may be divided among multiple diagrams simply to keep each diagram a reasonable size.

Many design tools and IDEs (integrated development environments) use interfaces that include a large tree widget that presents a “table of contents” of the design document and allow fairly direct access to any design element, regardless of how the design document is broken down into diagrams. These “table of contents” views support designers in finding individual design elements regardless of which diagram the elements reside in; however, standard “table of contents” views provide little help for visualizing semantic structures in the design.

Argo/UML augments these standard, task-independent views of the design with task specific ones. In particular, this subsection discusses the “navigational perspectives” cognitive support feature. Beyond providing a simple “table of contents,” Argo/UML’s navigational perspectives highlight tree structured relationships in the design document that may be difficult to understand from looking at design diagrams themselves. Argo/UML gives the designer a much richer set of alternative tree-structured views of the project, and provides a language for designers to customize those perspectives or add new ones.

Description. A designer using Argo/UML initially sees the package-centric navigational perspective but choose a new navigational perspective from the menu above

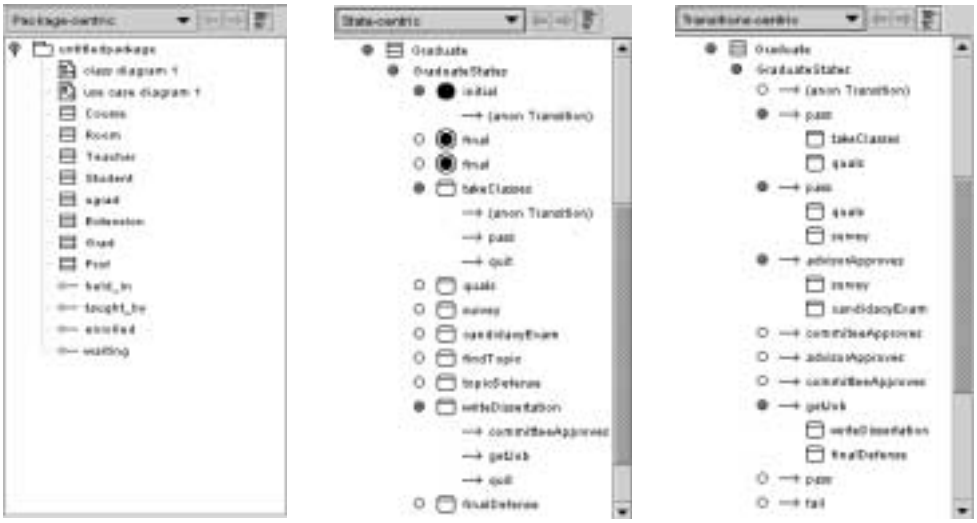

Figure 4-11. (a) “Package-centric” navigational perspective,
(b) “State-centric” navigational perspective,
(c) “Transition-centric” navigational perspective

the navigation tree (Figure 4-11a). For example, if the designer is working to define the possible states of a particular class, the state-centric navigational perspective shows states as the children of classes and state transitions as the children of states (Figure 4-11b). 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 navigational perspective shows the transitions as the children of the class and the states as the children of the transitions (Figure 4-11c).

Argo/UML contains several predefined navigational perspectives that support various tasks in object-oriented software design. For each of these tasks I have identified questions about the design that the designer must answer during that task. One such perspective is the transitions-paths perspective: it shows initial states as the children of classes and successor states as the children of states. This helps the designer answer the

Figure 4-12. Argo/UML's navigational perspective configuration window

question "if the object leaves this state, where can it go?" A related question is "how can the object get into this state?" Argo/UML does not provide a predefined perspective to answer this question, but the designer can use a configuration window to define new perspectives to answer new questions as they arise.

Argo/UML's navigational perspective configuration window is shown in Figure 4-12. The top pane lists currently defined perspectives. The lower left pane lists all predefined navigation rules, while the 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 "Class->Initial States" will be applied to any tree node that represents a class and will generate one tree node for each initial state in the state machine for that class. The set of possible navigation rules is large but finite; the UML standard meta-model (i.e., design representation) includes about 100 associations, each of which can have a corresponding

navigation rule. Navigational perspectives are generated by applying all applicable rules whenever a tree node is expanded by the user.

Mapping to theory. The cognitive theory of comprehension and problem solving indicates that designers need visualizations that are specific to the design task they are working on. This applies to both the diagrams and other design visualizations, such as the navigation tree and table views (described above).

When designers do not have design views that present all the elements and relationships needed for a given design task, they must combine elements from different design views to build a mental structure suitable for the task. Building such a structure in short-term memory (STM) requires mental effort and uses up STM resources needed for storing task plans and relevant knowledge. Mental visualization of complex structures can be error-prone if items are lost from STM. In fact, if the designer does a moderate amount of work on each element of the design element structure in his or her STM, elements are likely to be "pushed out" as STM resources are used in considering the first few elements.

For example, consider a designer who is checking that each class in an object-oriented design has a "save to disk" method that properly takes into account all composite and aggregate classes. The is-part-of hierarchy of a complex system is likely to involve elements and relationships from several class diagrams. The designer is further burdened by the need to determine whether each aggregate class should be saved as part of another class (by value) or independently (by reference). Since keeping a large is-part-of hierarchy in STM taxes limited resources, the designer is likely to need to refresh memory

by repeatedly scanning all the diagrams involved. Such repeated scanning is itself a time-consuming and error-prone process.

In contrast, Argo/UML's predefined navigational perspective for aggregation automatically constructs and presents the structure in question. The designer can simply expand the tree structure to the desired level of detail and progress systematically, line-by-line, with very little STM load.

Possible extensions. Navigational perspectives are an exciting and potentially very useful feature. Rather than pursue this particular feature as far as possible, I have opted to explore a broad set of proposed features. A possible extension to this dissertation would be to implement and evaluate the following enhancements to Argo/UML's navigational perspectives. First, the navigation pane could be split to show two trees with the second tree rooted at the selected node in the main tree. This might allow designers to see details of part of the tree without losing sight of the context of the parent node. Alternatively, the lower half of the navigation pane might show a list of the most-recently-used design elements to aid designers in flipping back and forth between two elements when comparing them. Also, the composition of navigational perspectives from rules might be made more powerful by allowing hidden tree levels, i.e., levels that are traversed on the way to the desired elements but that are not shown themselves. Finally, on-line documentation describing each perspective is likely to greatly help designers in selecting the proper perspective for their task, as would a task-oriented listing of available perspectives and rules.

4.3.2 The Broom Alignment Tool

Background. Designers typically prefer diagrams that look neat and orderly. Careful alignment of diagram elements can also serve as a form of secondary notation, as discussed below. Most design tools provide features to help align diagram elements with each other. Such features include nudging, grids, gravity, constrained movement, alignment commands, and guidelines. Nudging allows precise manual positioning of diagram elements. Alignment grids are found in virtually all drawing tools, and are used to restrict movement of elements to evenly spaced coordinates. Grids help align objects with each other by aligning each of them with invisible grid lines. Gravity causes an element being moved to snap to nearby stationary elements. Computer Aided Design (CAD) tools and high-end illustration programs often define gravity points at the corners, centers, and other geometrically important points on shapes. Gravity mainly helps establish adjacency (i.e., touching) relationships, but it can also be a step in aligning objects. Horizontally or vertically constrained movement allows users to interactively change one coordinate without accidentally changing the other. For example, if the tops of two objects are aligned using gravity, a constrained movement command can separate them horizontally while keeping the tops aligned. Alignment commands are found in the menus and toolbars of most drawing tools (e.g., align tops, align left edges, align centers). These commands explicitly move the selected objects into alignment with each other. Lastly, guidelines are typically found in page layout tools and are non-printing diagram elements that cause other elements to snap to them. Argo/UML provides nudging, grids, alignment commands, and a novel “broom” tool.

Each of the standard alignment features has its own disadvantages. For example, alignment commands demand that designers select the objects to be aligned then issue an alignment command. Selecting objects may require multiple selection actions. These selection actions may accidentally change the spacing between objects, and in some UML tools it may accidentally activate editing of the labels on a target object. Issuing an alignment command requires designers to imagine the resulting positions of the objects based on command names and icons. Some tools provide an alignment dialog box with a preview function but this introduces additional modality into the user interface. Argo/UML provides these standard alignment features and tried to mitigate their disadvantages (e.g., label editing is disabled if multiple objects are selected). Moreover, Argo/UML also provides a novel alignment feature that avoids these disadvantages.

Argo/UML's broom alignment tool is specialized to support the needs of designers in achieving the kind of alignment used in UML diagrams. It is common for designers to roughly align objects as they are created or by using simple movement commands. The broom is an easy way to precisely align objects that are already roughly aligned. The broom also takes advantage of the fact that, in design diagrams, objects are typically aligned along the X- or Y-axis rather than along arbitrary, diagonal lines. Furthermore, the broom's distribution options are suited to the observed needs of UML designers: making related objects appear evenly spaced, packing objects to save diagram space, and spreading objects out to make room for new objects. The broom also makes it easy to change from horizontal to vertical alignment or from left-alignment to right-alignment. While the broom is useful in a UML design environment, several of its aspects may also suit the needs of diagramming tasks in other design domains.

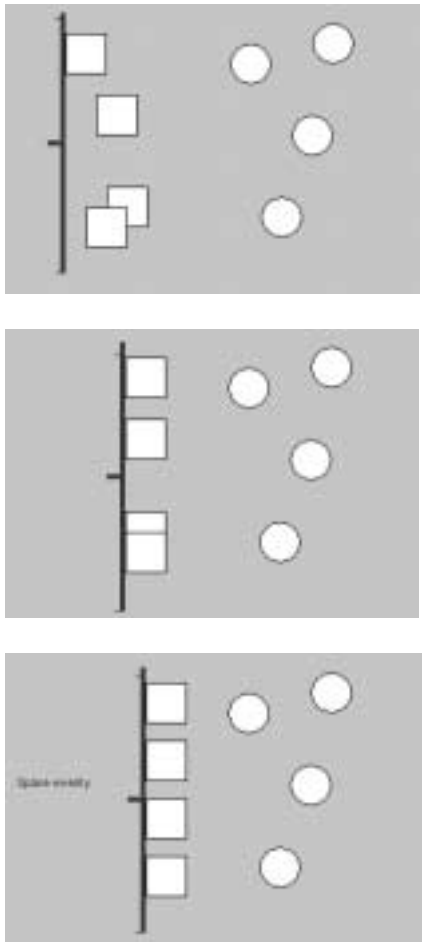

Figure 4-13. Aligning and distributing objects with the broom

Description. The T-shaped icon in Argo/UML's diagram toolbar invokes the broom alignment tool. When the mouse button is pressed while in broom-mode, the designer's initial mouse movement orients the broom to face in one of four directions: north, south, east, or west. After that, mouse drag events cause the broom to advance in the chosen direction, withdraw, or grow in a lateral direction. Like a real-world push broom, the broom tool pushes diagram elements that come in contact with it. This has the effect of aligning objects along the face of the broom and provides immediate visual feedback (see Figure 4-13). Unlike, a real-world broom, moving backwards allows diagram elements to

return to their original position. Growing the broom makes it possible to align objects that are not near each other. When the mouse button is released, the broom disappears and the moved objects are selected to make it easy to manipulate them further.

If the designer presses the space bar while using the broom, objects on the face of the broom are distributed (i.e., spaced evenly). Argo/UML's broom supports three distribution modes: objects can be spaced evenly across the space that they use, objects can be packed together with only a small gap between them, or objects can be distributed evenly over the entire length of the broom's face. Repeatedly pressing the space bar cycles among these three distribution modes and displays a brief message indicating the operation just performed (see Figure 4-13).

The fact that the broom pushes objects that it touches relieves the designer of the need to select target objects. This reduces the number of mouse movements needed and avoids accidental movement or editing of target objects. Argo/UML further reduces mouse movement by allowing users to invoke the broom by a control-drag rather than by using the toolbar button. Since objects are moved interactively, designers can see and judge the results of their actions immediately, without the need to interpret geometric terms (e.g., "align left edges"). The fact that objects return to their original positions when the broom withdraws allows designers to quickly undo undesired movements. Cycling through distribution commands also gives immediate visual feedback and reduces the need for designers to interpret geometric terms.

Mapping to theory. Alignment and spacing are important aspects of the secondary notation of design diagrams. Although the precise positions of diagram elements on the

screen do not hold formal semantic meaning in UML, it is a powerful visual cue that humans follow when reading UML diagrams. For example, the alignment and spacing of states in Figure 2-1 implies a temporal correspondence between certain parts of two parallel state machines, although the formal syntax of UML does not provide any formal means of representing or conveying that relationship.

Overall, the straightforward physical analogy between the broom alignment tool and real-world push brooms aids designers in understanding and anticipating the results of their actions. Shneiderman finds that users delight in using tools that provide "visibility of the objects and actions of interest; rapid, reversible, incremental actions; and replacement of complex command-language syntax by direct manipulation" (Shneiderman, 1998). At the cognitive level, users may find interfaces with Shneiderman's characteristics to require less planning, thus preserving short-term memory resources for task-level plans and relevant domain knowledge.

Alignment and spacing are two of the most practical visual aspects of a UML diagram to use for secondary notation. They do not rely on color, which may be difficult to print. They allow the sizes of diagram elements to be small, thus saving space. Alignment can clearly represent two informal aspects of the design: in Figure 2-1, X-axis alignment implies temporal correspondence while Y-axis alignment implies group membership. Alignment and spacing of diagram elements in a group is visually localized and thus reusable in different parts of the diagram for different meanings: ten groups of aligned objects on one page is readable, whereas ten color-coded or size-coded groups would be difficult to read.

Possible extensions. The broom mode can be enhanced by adding new types of alignment and distribution that are appropriate for a given diagram type. For example, pushing a class hierarchy upward might align classes in the bottommost row of the hierarchy along the face of the broom, but push superclasses in advance of the broom face so as to maintain the tree shape of the hierarchy. Also, the broom could be improved by using information about the connectivity of diagram nodes when breaking distribution ties. For example, if a set of horizontally aligned states is pushed leftward until they are all on top of each other and then spread out by pressing the spacebar, the resulting order of states is essentially random since they all had the same coordinates at the time of the distribution. This may cause many state transitions to cross each other. Connectivity information could help select an ordering that avoids edge crossings.

Related work. Raisamo and Raiha (1996) describe an alignment feature called the alignment stick that uses the same basic metaphor as Argo/UML's broom. However, Argo/UML's broom is unique in its ability to easily undo accidental movements and its support for both alignment and even spacing. Also, the intent of the alignment stick research was to explore the use of two-handed input mechanisms: a mouse is used to control the position of the stick, while a trackball is used to control the length and orientation of the stick. In contrast to the UML diagram focus of Argo/UML's broom, the alignment stick is not domain-specific and does not emphasize rectilinear alignment over diagonal alignment. Raisamo (1999) later extended his work on the alignment stick to produce several other interactive tools related to artistic drawing, for example, the carving stick removes some of the area of shapes when it touches them.

4.3.3 Model-based Layout

Background. Many current CASE tools provide a feature to automatically lay out diagram elements. Dozens of layout algorithms have been devised (e.g., Bertolazzi, Di Battista, and Liotta, 1995). The layout algorithms are typically domain-independent and seek to optimize domain-independent metrics, such as reducing the number of line crossings. Some layout algorithms can be customized with constraints (e.g., Graf and Neurohr, 1995; Ryall, Marks, and Shieber, 1997) that can be used to maintain domain-specific layout conventions. For example, class diagrams in UML are usually drawn with superclasses above subclasses.

The design process is made up of episodes in which the designer addresses different aspects of the design. Within each episode, task-specific questions about the design must be answered. For example, when considering an error condition in which a state machine should transition to a fail-safe state, the designer must answer the question as to which states can give rise to the error.

The domain-independent layouts provided by current tools do not support designers in answering task-specific questions. Designers must visually scan diagrams looking for elements involved in their current task. Continuing the example above, the designer must look at each state in the state diagram, recall the meaning of that state, and consider whether the error could arise in that context. This kind of visual scanning is somewhat error-prone because the designer may skip a diagram element accidentally, especially if the short-term memory load of evaluating a given element is high. Furthermore, since the scan order is not correlated with the semantic properties of the elements, considering

successive diagram elements may require different parts of the designer's knowledge to be activated, leading to a short-term memory effect analogous to thrashing in computer memories. As noted in Section 2.2.2, high short-term memory loads can induce procedural errors, namely super-goal kill-off, which can cause the designer to fail to completely scan the diagram.

Description. Model-based layout is a proposed feature for Argo/UML that is intended to help designers answer task-specific questions about the design. It is an automated layout feature which makes use of standard layout algorithms, but adds further constraints that position diagram nodes in diagram regions based on semantic properties relevant to the task at hand.

Figure 4-14a shows an example of a state machine diagram for an alarm clock as it might appear after a standard automated layout. Each state is shown with its state invariant condition. If the designer wanted to check each state in which the alarm was ringing, he or she would have to visually scan each state in the diagram and interpret the meaning of each invariant. Verifying completeness of non-existence properties, e.g., that no state allows both ringing and snoozing, always requires a complete scan. Furthermore, designers are likely to mentally factor the search criteria to simplify initial scans and come back to candidate states. For example, a designer might first scan for all states where ringing is allowed and then reconsider them in terms of whether snoozing is allowed. When using this strategy, short-term memory loads and the associated risk of skipping important elements increases with the complexity of the search criteria.

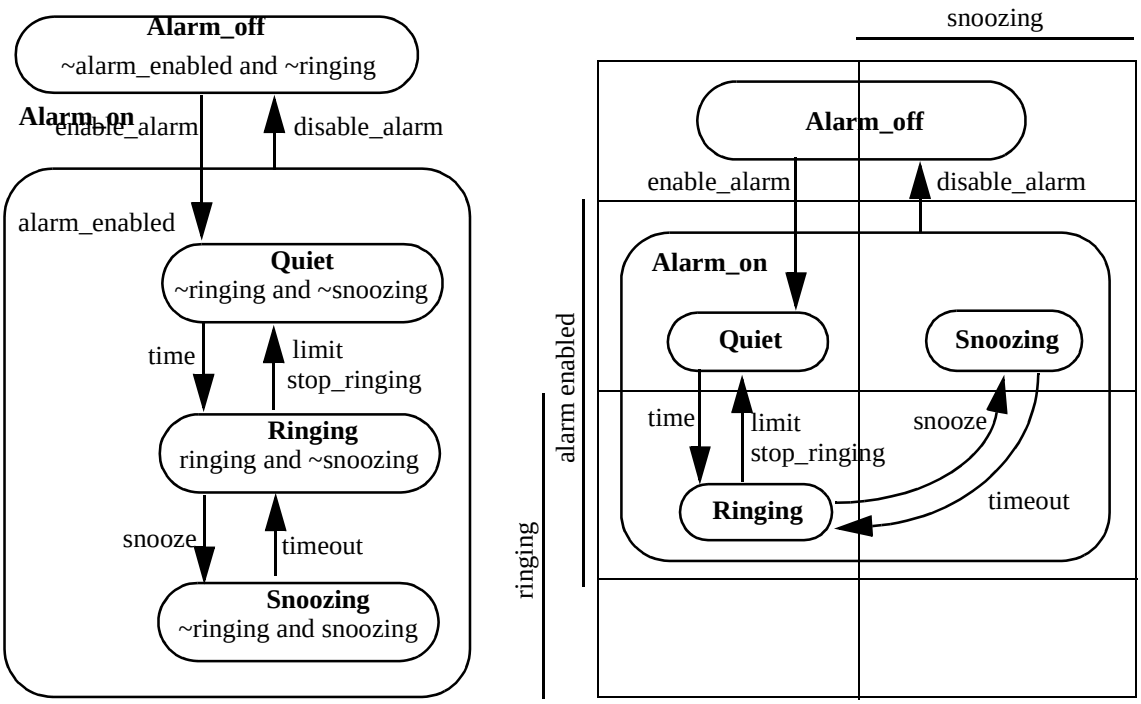

Figure 4-14. (a) Standard automated layout of a state diagram,
(b) Model-based layout of a state diagram

Figure 4-14b shows how the same state diagram would appear after model-based layout. Each row and column is labeled with the name of one of the variables used in the state invariants or the negation of that variable. The intersection of each row and column defines a rectangular region. States are constrained to appear within all appropriate regions. As long as the region constraints are satisfied, standard layout algorithms are used to improve readability by, e.g., reducing edge crossings. In this example, the rows and columns are arranged as they would be in a Karnaugh map (Ercegovac and Lang, 1985). To answer questions about states in which the clock is both ringing and snoozing, the designer need only scan states within the ringing and snoozing region. Furthermore, in contrast to the global scanning approach, increasingly narrow search criteria tend to make scanning easier with model-based layout. Even if the designer must scan the entire

diagram for some reason, diagram nodes are grouped by relevant semantic properties, thus helping to reduce the potential for short-term memory thrashing.

Layouts constrained to the cells of Karnaugh-maps are possible only when the constraining model attributes are a small number of boolean variables or conditions. In the more general case, any diagram region can be associated with an arbitrary condition; conditions are joined with logical-and where these regions overlap; and diagram nodes that satisfy the condition for a given region are placed within that region. Diagram nodes that satisfy the conditions of two disjoint regions are shown in both of them, while nodes satisfying two adjacent regions are positioned to straddle the common boundary. Diagram nodes that do not satisfy any region constraint are placed outside of any region. For example, a UML class diagram could be arranged on a Venn diagram with three circles labeled with the names of three members of the unit testing team.

Designers specify how these layouts are constructed by using one of the tabs in the model-based layout window. The first tab contains on-line documentation for this feature. The second tab assumes a row and column layout and prompts the designer to enter the conditions for each row and column. Exact sizes are not specified; rather, they are determined by the nodes that fall within any given region. That is to say, a cell will be exactly wide enough to comfortably fit the nodes that logically belong there. The third tab (Figure 4-15) allows the designer to take more control by specifying the size and location of rectangular, circular, and polygonal regions and their conditions. In each tab, several predefined layout options are provided based on experience with object-oriented design and new layouts may be saved for later reuse. The automated layout algorithm will do the

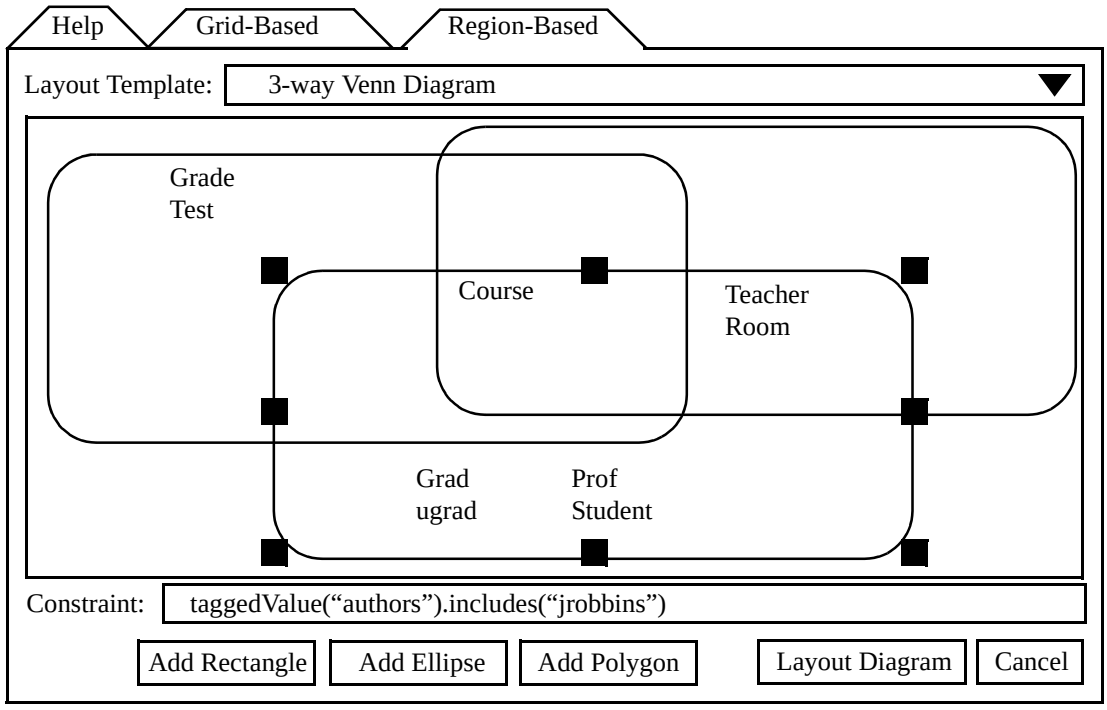

Figure 4-15. Configuring model-based layout with arbitrary constrained regions

best that it can to position nodes within these specified regions. If a region is too small for all the nodes that belong there, then some of the nodes will overlap others. Both tabs contain a preview of the resulting layout with a number shown in each region indicating how many nodes will be positioned there.

Once the designer indicates that all layout constraints have been specified then a new diagram is generated with the requested layout. In this diagram, each node may be manually moved, but only within the constraints of its logical region. Row and column layouts may be adjusted by changing the size of a row or column. In turn, this will cause nodes in subsequent rows and columns to shift.

Mapping to theory. The reasoning leading to this feature is described above. To sum up, model-based layout allows designers to more effectively answer task-specific

questions that arise during design. Designers working with diagrams arranged via model-based layout are expected to scan semantically defined regions rather than scanning the entire diagram. This is expected to allow them to answer task-specific questions more quickly, using fewer short-term memory resources, and with fewer oversights.

4.4 Construction Support Features

4.4.1 Selection-Action Buttons

Background. Argo/UML provides a toolbar of diagram elements that works very much like those found in other CASE tools. For example, the class diagram toolbar contains buttons for making new classes, new interfaces, new relationships between classes, and others. This is familiar to people who have used other CASE tools or drawing applications, but it is actually a rather poor interface for constructing structured diagrams because small targets that are far from the central work area are hard to select with the mouse. Furthermore, toolbar actions are typically too fine-grained and must be combined to achieve basic design manipulations. Although these difficulties with toolbars are straightforward, the ubiquitous use of toolbars may have prevented CASE tool developers from pursuing better interfaces. Argo/UML provides a new feature that builds on the familiar concept of a toolbar but avoids its disadvantages.

Description. When the user selects a node in a UML diagram, several handles are draw on it to indicate that it is selected and to provide user interface affordances to resize the node. Argo/UML also displays some “selection-action buttons” around the selected node. Figure 4-16 shows the handles and selection-action buttons on a UML class.

Figure 4-16. Selection-action buttons on a UML class, interface, and state

Selection-action buttons offer common operations on the selected object. For example, a class node has a button at 12-o'clock for adding a superclass, one at 6-o'clock for adding a subclass, and buttons at 3-o'clock and 9-o'clock for adding associations. These buttons support a "click or drag" interaction: a single click creates a new related class at a default position relative to the original class and creates a generalization or association; a drag from the button to an existing class creates only the generalization or association; and, a drag to an empty space in the diagram creates a new class at the mouse position and the generalization or association. Argo/UML provides some automated layout support so that clicking the subclass button three times will position the new classes so that they do not overlap.

Selection-action buttons are transparent. They have a visibly recognizable rectangular shape and size and they contain an icon that is the same as the icon used for the corresponding type of design element on the standard toolbar. However, these icons are unfilled line drawings with many transparent pixels. This allows selection-action buttons to be overlaid onto the drawing area without overly obscuring the diagram itself. Also, the buttons are only drawn when the mouse is over the selected node; if any part of the diagram is obscured, the mouse can simply be moved away to get a clearer view of the diagram.

Most CASE tool user interfaces are inspired by MacDraw-style drawing tools. These tools give the user total control over the order of their actions, but no support for ordering actions effectively. Typically, the toolbar buttons correspond to tiny actions, and it is up to the designer to form a plan that combines these tiny interface actions to achieve a task-level goal. For example, the designer might think, “I want a base class with a subclass.” The plan for doing that requires three normal toolbar clicks, two diagram clicks, and a drag. But if the designer wants a base class with five subclasses, he or she is likely to change the plan to reduce from eleven toolbar clicks to only two toolbar double-clicks (one to lock class mode and one to lock generalization mode). Expert users do this easily, but beginners have trouble and get distracted from the larger design task. With selection-action buttons, the tool “knows” that once you have a class it is likely that you will add a subclass. Adding subclasses can be done with as little as one selection-action button click per subclass, and there is much less need to plan or optimize user interface actions.

Mapping to theory. At the cognitive level, selection-action buttons are a real improvement over existing toolbars for two reasons.

First, Fitts’ Law basically says that the time it takes to move the hand or mouse from one region to another depends on the distance moved and the size of the target region. Current toolbars are very poor in this respect since they are frequently used, far away from the work area, and have small target regions. In contrast, selection-action buttons are located very near the location in the diagram where the mouse is positioned. Also, it is reasonable to have bigger buttons since they are translucent and do not require dedicated screen area.

Second, simpler user interface interactions distract less from the design task. Byrne and Bovair (1997) describe an experiment that showed that people who have other things on their mind are likely to make procedural errors in complex user interface tasks. In particular, they commit super-goal kill-off errors: i.e., they finish one difficult part of the task and then forget about their overall goal.

4.4.2 Create Multiple

Background. Creation of design elements is a task that occurs at the beginning of every design project and recurs often throughout the project. In fact, email conversations with several Argo/UML users have indicated that they encountered difficulties in constructing their design diagrams before they get the benefit of Argo/UML's other cognitive support features. The create multiple feature proposed in this section is a novel user interface feature that automates and informs low-level construction activities.

One form of domain knowledge that can be applied to aid designers in construction is design patterns. The basic idea of design patterns is that certain design fragments have been found to be frequently useful in a wide range of design problems (Gamma et al., 1995). If a given fragment is known to be frequently useful, then it is reasonable that a designer will want to include it in the design at hand. In addition to the recurring design fragment itself, a design pattern includes an explanation of why the fragment is useful and the situations where it is applicable. Making design patterns easy for designers to choose and apply is expected to reduce the overall knowledge burden of designers during construction tasks.

The user interfaces of most current CASE tools provide equal access to all features of the tool. For example, standard toolbars are a user interface element that provides equal access to all features and does not take advantage of the task-specific context or knowledge of the design domain. Also, most CASE tool user interfaces force the designer to frequently switch between keyboard and mouse input devices to alternatively place design elements and specify their names and properties. The create multiple feature helps designers specify design fragments by using just the keyboard to fill in a form. This is expected to reduce interaction time and effort for a common task as well as the feeling that arises of fighting with the tool when tools have difficult interfaces.

Description. "Create multiple" is a proposed Argo/UML feature that is intended to help designers easily create design fragments rather than individual design elements. These design fragments consist of multiple design elements and their relationships.

A designer opens the "Create Multiple" window via a menu command. The window consists of several tabs. The first tab offers on-line help on how to use this feature. The

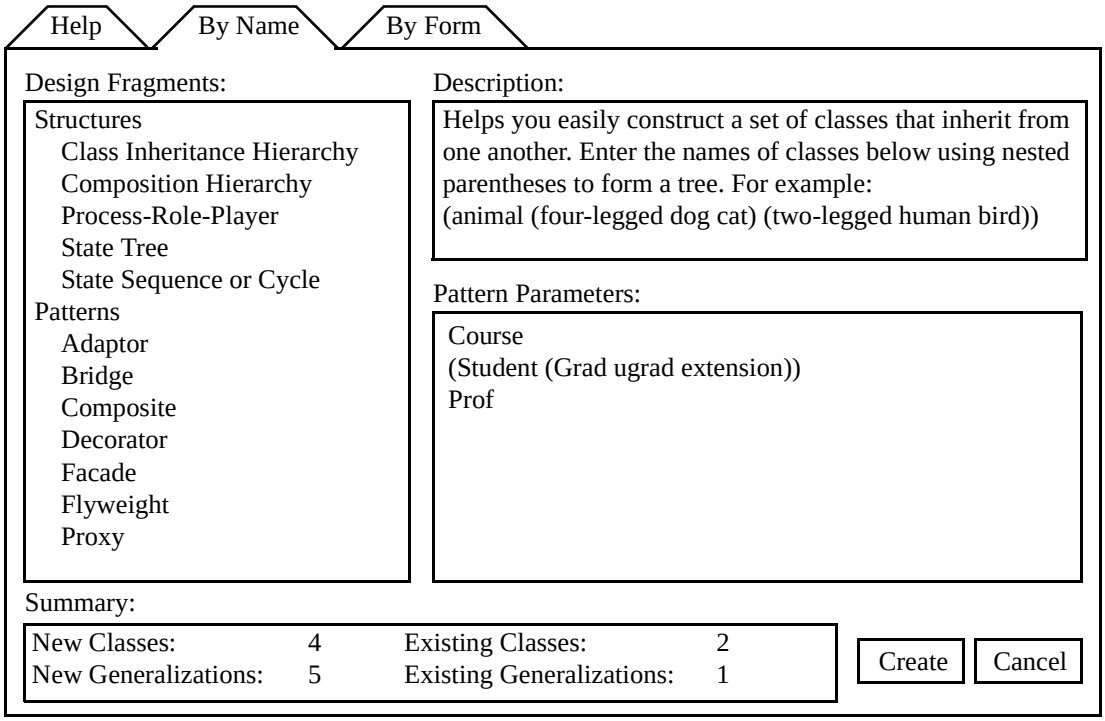

Figure 4-17. Mock-up of window to create multiple elements by pattern name

second tab, “By Name” (Figure 4-17), allows the designer to select a design fragment by name and to enter customization parameters. The set of design fragments offered by Argo/UML depends on the type of the current diagram. For example, if the designer is working on a class diagram and chooses “Class Inheritance Hierarchy”, the customization section will prompt the designer to enter the names of the classes in the hierarchy, and pressing the “Create” button will add the new classes and the proper inheritance relationships to the current diagram.

The third tab, “By Form”, support better visualization of specific design fragments and their combinations (Figure 4-18). In this tab, several design elements are shown in the context of one or more interwoven design fragments. For example, Figure 4-18 shows classes in both an inheritance hierarchy and a containment hierarchy, along with the

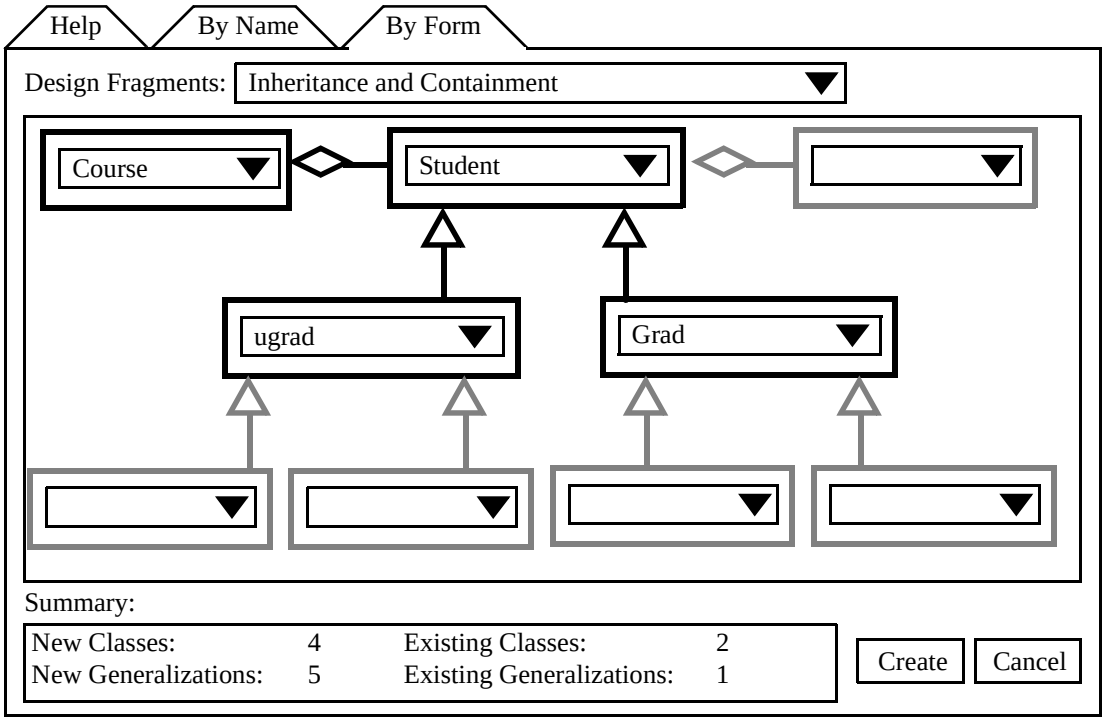

Figure 4-18. Mock-up for creating design fragments by form filling

composite pattern (Gamma et al., 1995). Although the tab shows a diagram, the position of the diagram elements cannot be changed and items cannot be removed or added. Instead, emphasis is placed on editing the names and properties of the available elements. Initially, all elements are shown with gray outlines and empty names. Whenever a name or other property is filled in, the element's color is changed to black. When the designer presses the "Create" button, all black elements are created and added to the current diagram. The names of design patterns are shown near the elements that participate in those patterns. Clicking on one of these names brings up on-line help describing the design pattern and explaining its applicability.

Regardless of which tab is used, the bottom of the "Create Multiple" window shows a summary of the elements that will be created when the designer presses the "Create"

button. For example, in Figure 4-18 the summary indicates that the designer has specified four new classes and two existing ones. Whenever a name in the “Create Multiple” window matches the name of an existing design element it is assumed that the existing element should be used rather than creating a new one. This allows the designer to graft new fragments onto the existing design. The name fields use combo boxes to aid the user in entering the names of existing design elements; the list of offered names is filtered to include only appropriate elements, and the names of any appropriate elements that were selected appear at the top of the list.

Some elements in the presented design fragment use check boxes to indicate whether they should be created or not. This allows for the creation of unnamed design elements. Also, some elements of the design fragment may be mutually exclusive alternatives. For example, in a fragment that could include a direct connection or a mediator class, a “Use Mediator” checkbox would insert a class in the middle of the diagram if checked; otherwise, the mediator class is shown in gray and black associations are drawn directly between the cooperating classes.

Mapping to theory. The “Create Multiple” feature is inspired by the theories of designers’ limited knowledge, reflection-in-action, associative memory, Fitts’ Law, and limited short-term memory. These are discussed, in order, below.

Design patterns are a form of knowledge that is specific to the domain of object-oriented software design. Patterns can also be identified in specific application domains. However, a given designer has limited knowledge and is unlikely to be aware of all the design patterns that are applicable to the design at hand. Even an expert designer may

have difficulty recalling known design patterns at times when they are applicable. Forms in the “Create Multiple” window prompt designers to consider applicable patterns in the normal course of construction. The links to on-line documentation provide further knowledge support. Several design support tools have been built to help detect or document design patterns (e.g., Seemann and von Gudenberg, 1998; Keller et al., 1999). However, none of these tools effectively use patterns to aid construction.

Some tools offer design template files as a form of knowledge support for construction. However, these design templates are difficult to combine with each other and with elements of the design at hand. The grafting behavior of the “Create Multiple” feature addresses the fact that designers will not know what design patterns are appropriate until they have partially specified the design. The need to make design decisions in the context of a partially completed design rather than at the start of the design process is identified by the theory of reflection-in-action.

When design patterns are offered, they provide a visual cue to the designer to consider how that pattern might fit into the design at hand. For example, when the designer sees the composite design pattern offered, he or she is cued to recall or imagine the composite relationships that belong in the design at hand. Each element with an empty name is a prompt for the designer to think of candidates for that position in the design fragment.

Since the “Create Multiple” window consists of standard widgets for text entry, it can be operated entirely via the keyboard rather than the mouse. This provides an important alternative to standard diagram creation features that are heavily mouse-oriented. Most software designers are experienced typists and are expected to appreciate the keyboard-

only option. Since the designer may select target fields by using the tab key, rather than the mouse, the size and location of the fields is not so critical and small elements can be used. This is in contrast to mouse-centric editing interfaces in which smaller target areas increase selection time as described in Fitts' Law (Fitts, 1954).

Furthermore, since the positions of nodes cannot be changed in the "Create Multiple" window, there is no need to provide distinct user interface modes for diagram editing and property editing. Instead, a single, familiar form-filling interaction mode is used. It is expected that using a "mode-less" interface will reduce the need for the designer to plan his or her interactions with the tool, and thus will keep more short-term memory resources available for task-level planning.

Possible extensions. One possible extension to this feature would be to build more knowledge about the applicability of each design pattern into the tool itself. This could be done by augmenting each pattern with a guard condition that would disable patterns that are inappropriate for the design goals or the initially selected design elements. Those patterns that are appropriate could be grouped or automatically ranked according to their suitability for achieving specific design goals.

Two other possible extensions would be to supply wizards that prompt the designer for additional pattern creation parameters as needed, and to define critics that are related to the pattern as a whole rather than the individual design elements that comprise it.

Related work. The literature on design patterns has focused on collecting patterns into pattern catalogs (e.g., Gamma et al., 1995). Ideally, the resulting collections define a

“pattern language” consisting of the key design patterns for a given domain (Alexander et al., 1977). This approach serves to document the patterns used in a given design community, and it builds agreement among those designers who choose to study the pattern language, but it does not strongly support designers in applying design patterns. To be more effectively used, design patterns must be more readily available to designers at the time when they are needed. Also, the pattern language approach requires that designers learn the patterns before they are needed. In contrast, the “Create Multiple” feature offers design patterns directly in the tool that designers use to build the design, and it offers designers the ability to choose from the visually offered patterns rather than recall one by name.

Coad and colleagues (1999) proposes a set of design patterns for business applications. Coad has built support for these design patterns and others into the Together/J UML tool. Constructive pattern support in Together/J is similar to the “By Name” method of pattern selection proposed for Argo/UML. In each feature, the tool contains knowledge about available design patterns and offers them to the designer. However, Together/J presents only the names of each design pattern rather than offering them visually. This helps automate the design construction task, but it assumes that the designer has fairly complete knowledge of the available patterns and their applicability. In contrast, Argo/UML’s form filling method offers design patterns visually, in a way that prompts designers with potentially useful design fragments without requiring foreknowledge of a pattern language. Furthermore, Together/J grafts new elements into the design in a preprogrammed way that cannot be specified by the designer. In contrast, Argo/UML considers grafting as a key part of pattern application.

4.4.3 Visual Blender

Background. Design is a creative activity. Creativity in object-oriented design need not be the bold creativity found in the arts or the paradigm shifts needed for major scientific discoveries. It is simply the daily creation of solutions to design problems.

Designers often come up with new ideas for solutions while brainstorming with other stakeholders using a whiteboard. Whiteboards fit the Geneplore theory in that they allow for generation of ideas without evaluation. However, whiteboards are passive and do not help designers who need help in being creative. Discussing designs with other stakeholders can lead to novel combinations of ideas that neither person would have thought of alone. However, such meetings require at least two designers to work together, and they might be made more effective if the participants brought more creative ideas that they generated independently before the meeting.

Description. Argo/UML's proposed visual blender feature is intended to help individual designers generate creative ideas by exploring combinations of existing ideas. It does this by presenting visual images that emphasize the combination or interaction of design concepts. The various parts of each image are labeled with terms taken from the partially specified design or a list of terms entered by the designer.

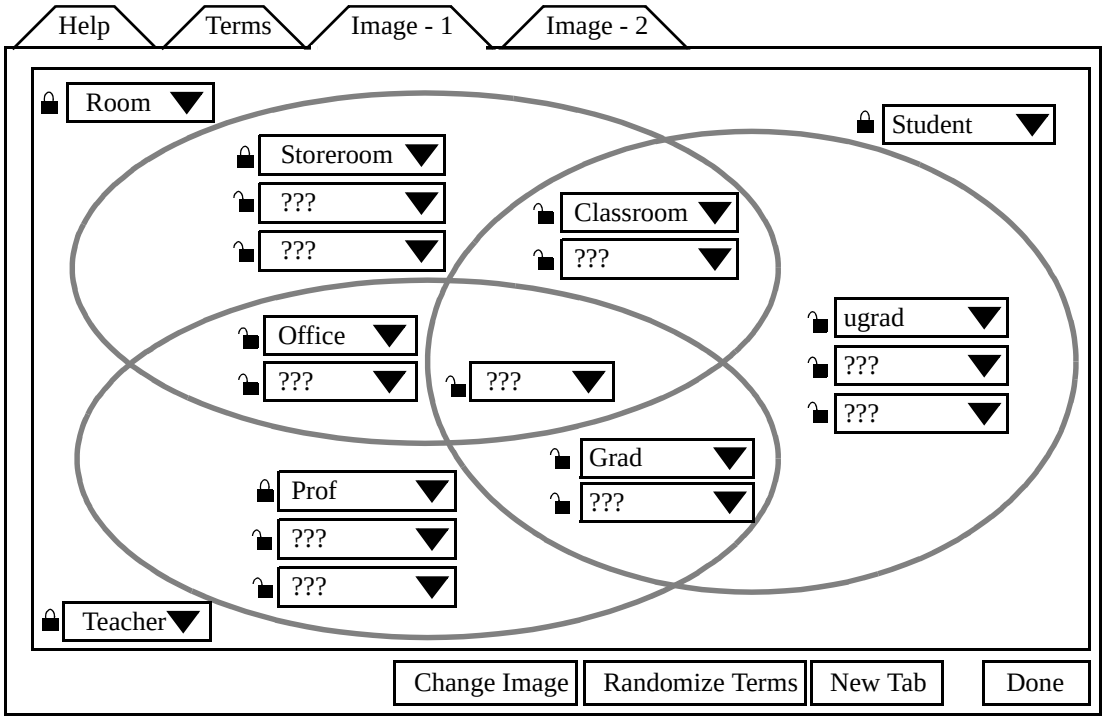

Figure 4-19. Mock-up of the visual blender window

Designers activate the visual blender window (Figure 4-19) via a menu command. The window consists of several tabs. The first tab contains on-line documentation for this feature. The second tab lists a set of terms to display on the images. These terms are initially taken from the names of elements in the current design, but the designer is free to edit the list. The third tab and any additional tabs show the images. The “New Tab” button inserts a new tab where new images will be shown and shifts existing tabs to the right. Inserting tabs provides a convenient way to save and access images that the designer might want to review later.

Each image contains a predetermined set of label locations that can be filled in with terms from the term list or other text entered by the designer. Each label is displayed in a

text field that allows the designer to manually change the term to another from the term list or enter new text. Each label can also be locked to prevent accidental change. The "Change Image" button replaces the image on the current tab with a new image from the image library. Changing the image will automatically add some random terms from the list if the new image has more predefined locations that the previous one. The "Randomize Terms" button will replace all unlocked terms in the current image with new terms selected randomly from the term list. As the terms are randomized, they are briefly animated in a way that suggests the rotating wheels of a slot machine. The text field at the bottom of the window is intended for the designer's notes and is shared among all tabs.

Mapping to theory. The proposed visual blender feature is primarily inspired by the Geneplore theory of creativity and, specifically, the visual combination method of generating new ideas. The cognitive theories of associate memory and fixation also play a role. The end result is a feature that is fairly similar to the suggested exercises for curing writers' block discussed in Section 2.1.3.

The visual blender feature uses a window that is independent of the main Argo/UML window to emphasize that the designer is free to generate new ideas without the constraints that normally apply. In the visual blender window, design critics are not active and the formal UML syntax is not enforced.

The images displayed in the visual blender window are intended to suggest a wide range of logical relationships between the items labeled with design terms. These images serve as cues for the designer to recall or imagine the ways that the suggested relationships apply to the design at hand. The images themselves are drawn as rough

sketches to avoid suggesting vivid colors or forms that may cue the recall of distracting memories of real world people or objects. Instead, the design terms themselves are the most visually compelling elements of each image, and are expected to help designers focus on the concepts and relationships of the design at hand.

The potential for cognitive fixation should be considered when presenting the designer with cues that take the form of examples or suggestions. As discussed in Section 2.2.3, designers are likely to fixate on examples that seem appropriate, even if they are not in fact appropriate. The visual blender feature mitigates the risk of fixation in two ways: the images presented are visually rough to imply that they are not finished examples, and the selection of design terms is done in an obviously random way to avoid any assumption that they are prepared examples.

Several aspects of the visual blender feature are also found in the suggested cures for writers' block. In both cases, the activity is an exercise that takes place outside the normal constraints of the design task. Also, both activities employ obvious randomness to help separate them from the main design task and to mitigate the risk of fixation. However, the proposed visual blender feature differs from the writers' block exercises in that the terms used in the visual blender images are drawn from the partially specified design rather than being completely random.

CHAPTER 5: Usage Scenario

The goals of this chapter are to demonstrate that Argo/UML is a useful and usable object-oriented design tool and to demonstrate the use of its cognitive support features in the context of a plausible design session.

Design of any real-world software system is a long-term process consisting of many hours of work spread over several design sessions. Opportunities for different kinds of cognitive support are distributed over the entire design process. The scenario in this chapter is composed of five scenes, each demonstrates how cognitive support may be provided at different stages of design. Each scene is presented using a condensed timeline to quickly reach the point of cognitive support; actual usage is not expected to be so densely packed with cognitive support opportunities.

The example design problem addressed by the designer in this scenario is the design of a university telephone enrollment system. This is a fairly standard design exercise that has been used in introductory books on UML (e.g., Quatrani, 1998; Booch, Rumbaugh, and Jacobson, 1999).

This scenario assumes that the designer is already familiar with the application domain and object-oriented design techniques, and that he or she has had some experience with Argo/UML. Furthermore, it is assumed that he or she has a general solution approach in mind.

5.1 Scene 1: Initial Construction, Error Detection, and Correction

Setting. In this scene, the designer dives right into construction of the design by placing design elements into a design diagram. Argo/UML supports construction in many of the same ways as standard CASE tools, but provides extra support for construction as well as error detection, error correction, and reminding.

Steps. Figure 5-1 shows the screen that a designer first sees after starting Argo/UML. Argo/UML provides the standard type of CASE tool interaction that object-oriented designers are familiar with. The designer can place design elements into a diagram by clicking on a toolbar button and then clicking in the diagram. Relationships can be defined by selecting the type of relationship from the toolbar and then dragging from one design element to another.

The designer knows that classes will be needed to represent students, classes, and rooms. These are easily created via the toolbar buttons. After the designer names one of

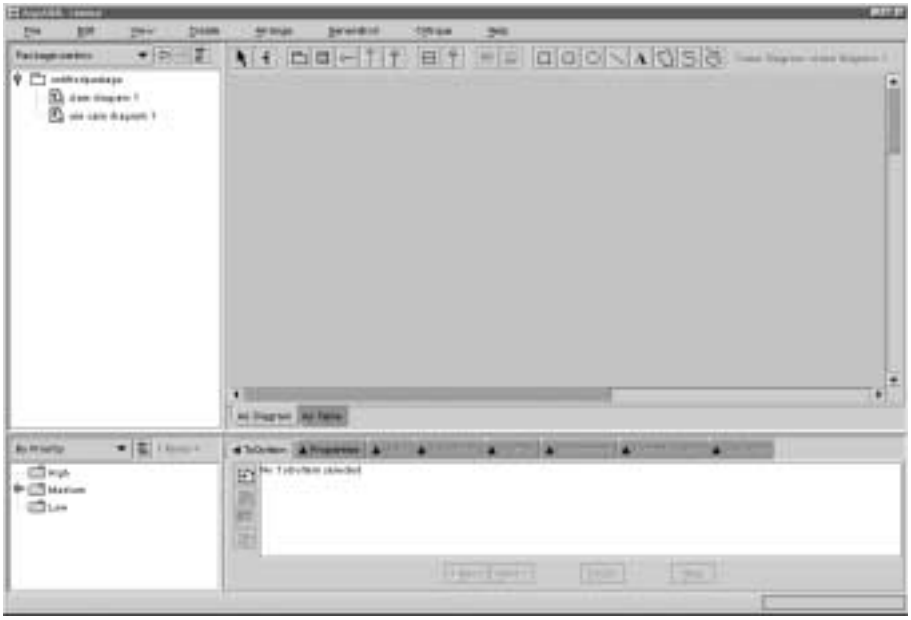

Figure 5-1. Argo/UML initial screen

the new class icons “Class”, a wavy red underline clarifier is shown under the name. The designer moves the mouse pointer over the wavy red underline and sees a small pop-up window stating “Change Class to a Non-Reserved Word.” In this case, the headline of the detected problem is enough to prompt the designer to change the name to “Course”. The designer names the other classes “Student” and “Room”.

Additional wavy red underlines are shown in the other compartments of each class icon and a yellow PostIt note appears at the top left of each class. The designer moves the mouse cursor to check out these other errors and sees, for example, “Add Attributes to Student” and “Add Associations to Student.” The designer recognizes both of these as prompts to specify additional aspects of the model.

The designer uses the familiar toolbar button style of interaction to define associations between Course and Room to model course locations. Then he or she adds an association from Course to Student to model course enrollment. Building the model to this point has required a fair amount of clicking and dragging, so the designer decides to use the selection-action buttons on Course to define the waiting list relationship between Course and Student.

At this point, the designer recalls the requirement that three types of students need to be treated differently, so he or she uses the subclass selection-action button on Student to define three subclasses with three mouse clicks. After the new classes are named “ugrad”, “Grad”, and “Extension”, the design appears as shown in Figure 5-2.

The designer continues to build the model by creating a new class for professors. Since using selection-action buttons seems easier than using the toolbar, the designer decides to drag from the association selection-action button on Course to create a new class and then names it “Prof”. Then, the designer recalls the requirement that graduate students may also teach classes and renames the current class “Teacher”. Next, the designer clicks the subclass selection-action button on class Teacher to create a new subclass for professors and the designer drags on the subclass selection-action button to make a new generalization edge to the existing Grad class.

At this point, the designer feels that substantial work has been done quickly and that it is now time to fill in some of the blanks that were left. While class Grad is selected, several clarifiers are shown, prompting the designer to point at them with the mouse. Pointing at the yellow PostIt note in the upper left of the Grad class produces a pop-up window with the text “Change Multiple Inheritance to Interfaces.” In this case, the prompt is not enough to tell this particular designer what to do. So he or she uses a pop-up

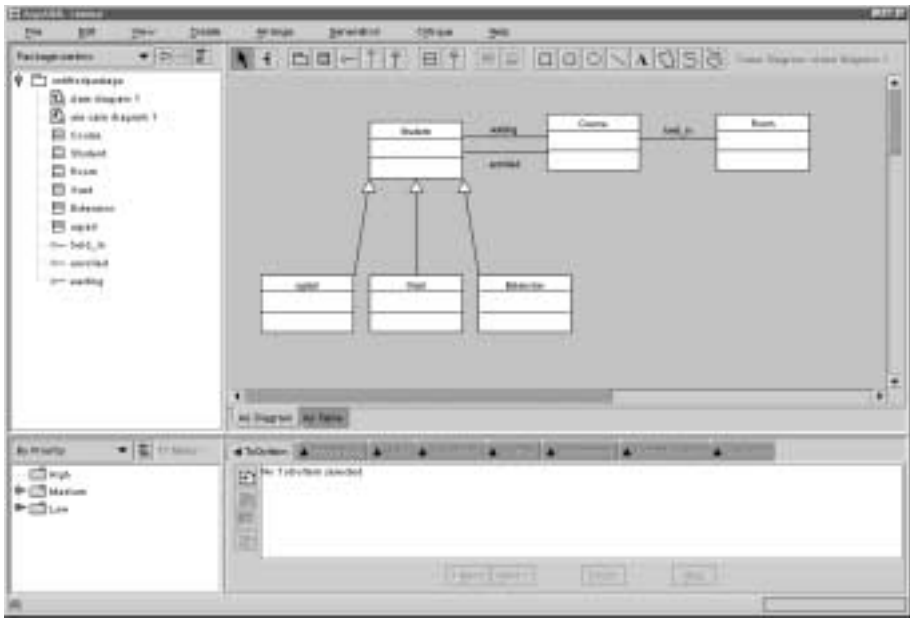

Figure 5-2. After placing initial classes

Grad has multiple base classes, but Java does not support multiple inheritance. You must use interfaces instead.

This change is required before you can generate Java code.

To fix this, use the “Next>” button, or manually (1) remove one of the base classes and then (2) optionally define a new interface with the same method declarations and (3) add it as an interface of Grad, and (4) move the method bodies from the old base class down into Grad.

Figure 5-3. “To do” item description

menu to get a list of all outstanding criticism on class Grad and selects the one labeled “Change Multiple Inheritance to Interfaces.” This item is the first item on the pop-up menu because the designer opened the pop-up menu with the mouse over its yellow PostIt note.

When the item is selected from the pop-up menu, the ToDoItem tab at the bottom section of the screen now changes to show the detailed description of the detected problem (see Figure 5-3). The designer does not understand the steps described in the third paragraph but thinks that the item is probably not relevant at this stage in the design process. He or she decides to investigate the provided wizard to see if resolving the item can be done easily. Pressing the “Next>” button at the bottom of the ToDoItem tab produces the first step of the wizard. The designer notices that the blue progress bar on the “Change Multiple Inheritance to Interfaces” item in the “to do” list has only moved a small amount, and decides that fixing this particular problem might require more learning and effort than he or she wants to spend on it at this time. Instead, the designer presses the “Add Item” button on the ToDoItem toolbar and enters a personal reminder to learn more about multiple inheritance in Java before it is time to implement the design.

Summary. In this scene, Argo/UML allowed the designer to directly manipulate the design in familiar ways. Meanwhile, Argo/UML provided knowledge support that helped detect errors and incompleteness in the design and guided the designer in resolving identified problems. Furthermore, selection-action buttons supported construction by providing a style of interaction that better matched the goal structure of the design construction task. In the final part of this scene, the designer added a reminder to Argo/UML's "to do" list to defer a decision for which he or she lacked needed knowledge.

5.2 Scene 2: Cleaning up the Design to Communicate Intent

Setting. Now the designer has placed a number of design elements and relationships in the design diagram. But the visual organization is unclear. In this scene, the designer cleans up the design diagram to more clearly communicate its intent.

In this design problem it is assumed that class Course is a central concept and its importance should be indicated as such in the design diagram. In contrast, it is assumed that rooms in this system are secondary. Likewise, the enrollment relationship is more fundamental to the design than is the waiting_list relationship. Furthermore, rooms must relate to facilities records that are supplied daily via a file transfer from a facilities management system, while course information can be edited at any time via an applet on a web page.

Steps. The designer moves class Course to be slightly above the center of the design diagram. This visually implies that class Course is a central concept of the design. It also

puts some of the classes too close together and makes some associations diagonal rather than rectilinear.

The designer then uses the mouse to move the other classes into their proper positions around class Course. Dragging the classes into position causes them to be somewhat misaligned with each other. The designer then uses the broom alignment tool to restore the visual groupings as shown in Figure 5-4. In particular, since the classes for undergraduate students, graduate students, and university extension students all fulfill the same role in the framework, the designer establishes a visual group for them. This is done by pushing them upward with the broom until they are horizontally aligned, then pressing the space bar to make them evenly spaced.

One convention of object-oriented design diagrams is that important relationships should be drawn with short, straight edges, while less important relationships are often drawn using bent edges. The designer accomplishes the straight edge effect for the

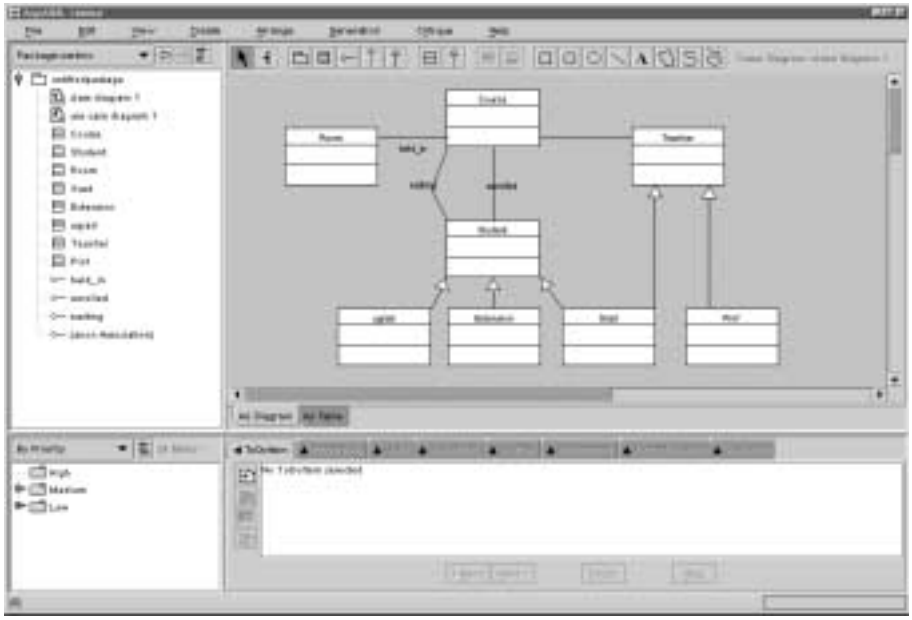

Figure 5-4. Reorganized class diagram

enrollment association by dragging on a point on the association to add a new vertex. When the vertex being dragged is near a node, such as class Course, the segment of the edge between the node and the mouse is automatically kept straight or rectilinear. The new vertex is automatically removed on mouse release because it is collinear with the endpoints of the edge. For the less important waiting_list association, the designer drags a point on the association to the left. The result is a two-segment angled association that appears secondary to the straight association.

Based on prior experience, the designer feels that the means by which data will be supplied will play an important role in upcoming design decisions. Since the UML notation does not define a way of documenting information sources, the designer uses two unstructured graphical elements to indicate these sources. A UML comment node could be used to textually describe the data source, but the graphical annotations shown in

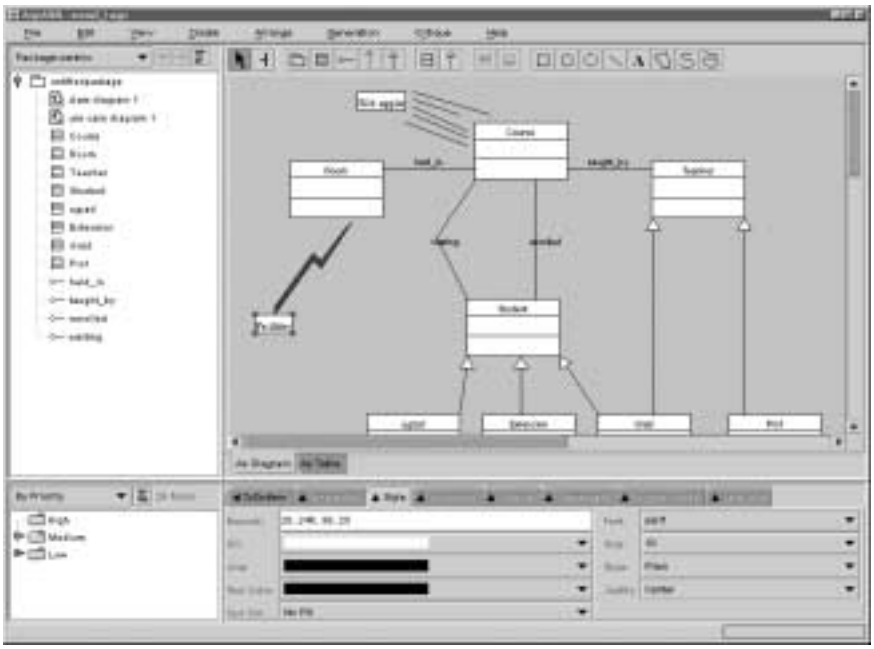

Figure 5-5. Class diagram with annotations describing data sources

Figure 5-5 illustrate the concurrency and data volume of each source in a way that will make a more immediate impression on the designer's colleagues.

Summary. In this scene, the designer used the broom alignment tool to help establish visual properties that emphasize some aspects of the design over others. This helped to communicate the intent of the design by adding new information in the form of secondary notation. In this case, the secondary notation was expressed with the position of elements on the page, alignment, spacing, and the straightness of edges. Informal annotations helped to add further information that was important to this design, even though it could not be expressed using the standard design notation.

5.3 Scene 3: Answering Questions that Arise During Design

Setting. As the design process progresses, a multitude of design decisions must be made. Each of these decisions must be informed by the designer's understanding of the problem domain, solution domain, and the state of the partially completed design. Designers formulate and reformulate their understanding of the state of the design as different design issues are addressed. Previous work on program comprehension has proposed a hypotheses-driven model in which programmers alternatively form questions and answer those questions (Brooks, 1983; Koenemann and Robertson, 1991). This scene demonstrates some of the questions that can arise while working with a UML design and how Argo/UML's cognitive support features might support designers in answering them.

In this scene, the designer has been working on the design for some time and its complexity has grown to, say, fifty classes and fifty associations spread over several class

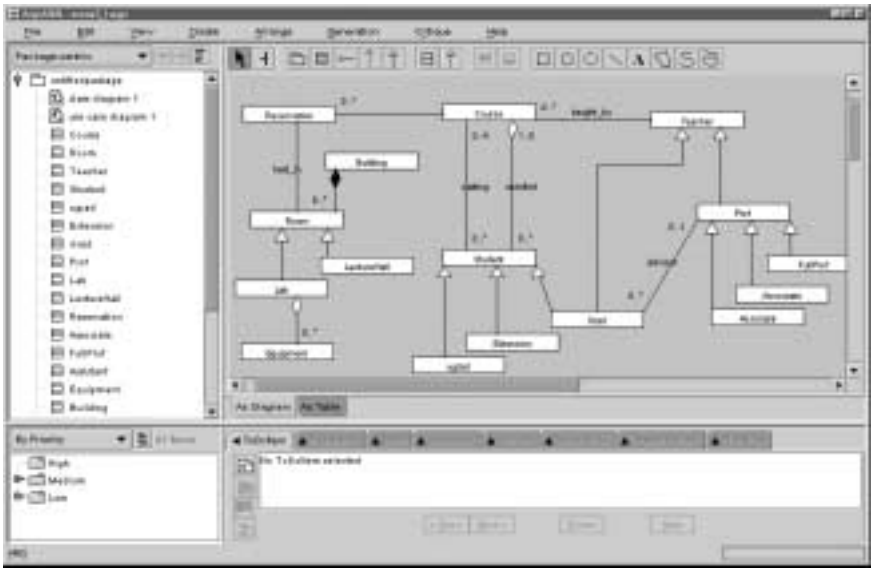

Figure 5-6. One class diagram of many after the design has grown

diagrams. Figure 5-6 shows one of these diagrams. At this stage in the design process, visually scanning design diagrams is no longer a reasonable way for the designer to answer his or her questions about the state of the design.

Steps. One of the critics offers the advice that the designer should consider combining two classes. The designer is puzzled as to why that advice is offered and looks at its detailed description. The critic suggests combining the Room and Reservation classes because instances of class Room will always be allocated in a one-to-one correspondence with instances of class Reservation. The designer realizes that the multiplicity of the held_in association is not specified correctly: each room should be able to have zero or more reservations. This prompts the designer to fix the problem at hand and then ask the broader question, “Does every association have the right multiplicity?”

Opportunistic table views help answer this question by presenting the design in a dense format that encourages systematic scanning. The designer clicks on the “As Table”

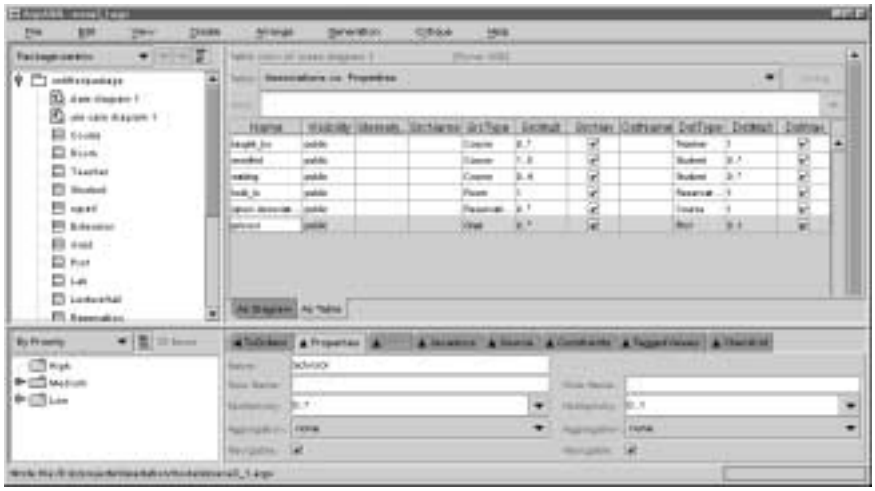

Figure 5-7. Table view of associations and their properties

tab at the bottom of the main editor window and selects the “Associations vs. Properties” table view (Figure 5-7). Using this view, the designer considers each row of the table by looking at the name of the association, its source multiplicity, and its destination multiplicity. As an aid to systematic scanning, the designer turns on Argo/UML’s row highlighting option and moves the selected cell down the table, row by row. Moving the selection causes each row to be highlighted in turn, thus helping the designer keep his or her place.

While scanning the associations, the designer finds that the enrollment association has the right multiplicity, but its aggregation is wrong. This prompts the designer to ask the question, “What about the aggregation of the other associations?”

To answer this question, the designer can simply scan down the table again, this time looking at a different column. The designer begins to do this visually, without moving the currently selected cell or changing the highlighted row. However, at some point the designer may need to refer to requirements or project-related email that is outside the

Argo/UML tool. If that happens the designer is likely to move the selected cell to the current row of interest to avoid losing his or her place. From there, systematic scanning can continue either visually or by continuing to use row highlighting as an aid. Once the designer completes the design excursion to satisfactorily answer the second question, he or she must return to the original question of multiplicity. Argo/UML's "instant replay" feature helps the designer recover from the design excursion by rapidly repeating the sequence of recently highlighted rows. This tells the designer how much progress was made on the first scan of the table and where to pick up again.

Eventually the designer feels that it is time to implement persistence. A natural question that arises in this recurring design task is "what are the aggregation hierarchies in the design?" This question is hard to answer if aggregation relationships are not emphasized in a diagram or if they are spread over multiple diagrams. Navigational perspectives help answer this question by presenting tree views of tree-structured relationships in the design. Aggregation of classes is one of the predefined navigational perspectives because it is known to be frequently useful in object-oriented design. The designer can simply select the "Aggregate Classes" item from the list of available perspectives above the navigation tree. This results in the tree view shown in Figure 5-8.

Another design issue that relates to aggregation hierarchies is how objects can be serialized and communicated between distributed components. In addition to understanding aggregation, the designer must ask "Which classes reside on both the client and the server?" Model-based layout is a proposed feature that could help to answer this question by reorganizing the layout of diagram elements but still presenting them in the

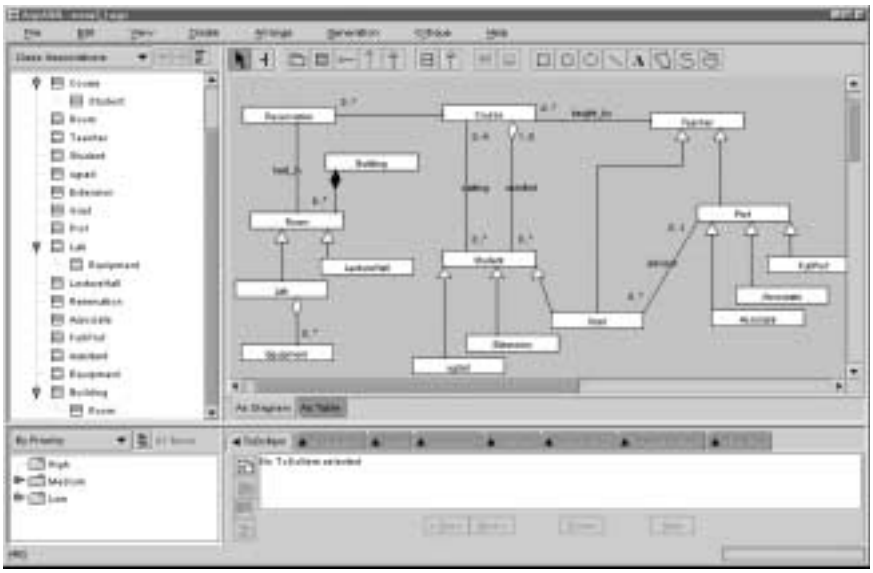

Figure 5-8. Aggregate classes navigational perspective

same context. For example, model-based layout could make the client/server division visually clear, while still showing associations between classes. It does this by automatically positioning elements while constraining them to appear in diagram regions based on model properties. In this example, the designer could choose the “Layout Diagram...” menu command and fill out a dialog box as described in Section 4.3.3. A mock-up of how the resulting diagram might look is shown in Figure 5-9.

Summary. In this scene, the designer asked several questions about the state of the partially completed design and answered them with the help of cognitive support features. Opportunistic table views helped the designer systematically scan the table and recover from certain kinds of design excursions. Then, navigational perspectives made important hierarchical structures in the design clearly evident. Finally, model-based layout helped make hidden properties of the design visible, while still maintaining the overall context of the design diagram.

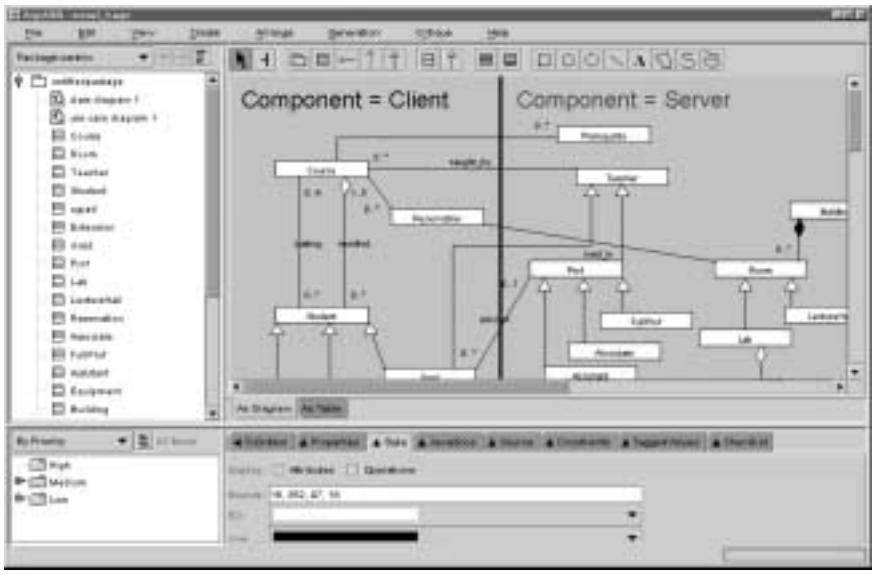

Figure 5-9. Mock-up of model-based layout

5.4 Scene 4: Considering the Important Issues

Setting. Sometimes designers don't ask the right questions about their design because of oversights or lack of expert knowledge. This scene consists of four short interactions that each show how Argo/UML might prompt the designer to work more like an expert. The emphasis here is on the ways the tool can communicate guidance to designers; the contents of the provided guidance is assumed to be insightful and informative, but it need not always be authoritatively correct.

Steps. Simplicity is one design quality valued by many object-oriented design experts. As the design document grows, expert designers tend to look for ways to simplify the design to make it more understandable, maintainable, and extensible. In contrast, novice designers are less able to recognize and combat unneeded design complexity. Unneeded complexity can become evident when the designer reflects on the partially completed design after having worked hands-on with the design and increased his or her understanding of the design issues. Revisiting the example used in previous scenes, the

designer will at some point notice the criticism of class Room. The suggestion to "Consider combining classes" prompts the designer to simplify the design where possible. In this case, the designer may change the multiplicity, as described above, or the designer could decide to simplify the design by having a single RoomReservation class. The actual decision should be based on the designer's understanding of the problem domain and the relative importance of different aspects of the system.

Another thing that differentiates expert from novice designers is that experts know the limits of their tools and languages and know where they break down and when to step beyond them. For example, when looking at the design checklist items provided for the cummulativeUnits attribute of class Student, the designer will see the checklist item "Does any other value need to be updated when this value is changed?" This prompts the designer to keep in mind the dependencies that are not represented in the standard UML design notation. Specifically, the designer is prompted to consider whether an attribute such as classStanding should be updated whenever cummulativeUnits is changed.

Expert designers also know to keep in mind how their task fits into the overall development process and their own role in the development organization. Organizational critics can prompt designers to consider issues that are important to the designer's organization, regardless of whether they are important to the designer. For example, one potential organization-specific critic might warn designers that incorporating untested components requires approval from the testing lead. This criticism cues the designer to consider how his or her design decisions will affect other project stakeholders, namely the testing staff. Taking the time to change the design or consult with the testing lead may not

be in the designer's short-term interests, but more accurate planning and coordination is in the overall interest of the development organization.

When expert designers work hands-on with a design, they tend to keep in mind a great number of relationships and design details that are then used to inform design decisions. Less experienced designers may have difficulty keeping in mind the same information or recognizing the value of recalling certain details when making future decisions. For example, when reviewing the design of the server side of the university enrollment system, the designer might use Argo/UML's search utility to find all classes that reside in server components[2]. If the designer selects the row for class Course in the main query result table, then class Prerequisite will also be listed in the related result table. Seeing class Prerequisite prompts the designer to recall the relationship between Course and Prerequisite, and to consider whether class Prerequisite should be on the client or server.

Summary. These four short interactions have shown how features of Argo/UML can help all designers work more like expert designers. Design critics and checklist items convey expert advice to designers engaged in the design process. In particular, organization-specific critics inform the designer of the implications of design decisions to the larger development process. Furthermore, Argo/UML's opportunistic search utility helps designers notice the same relationships that expert designers keep in mind when working with the design.

2. Argo/UML's search utility has not been implemented enough to perform the particular query in this example.

5.5 Scene 5: Resolving Open Issues Before Reaching a Milestone

Setting. The design process usually progresses in a very complex, iterative way; however, there are some clear milestones along the way. This final scene gives an example of how Argo/UML can support the designer in resolving open issues to reach a design milestone.

Steps. Eventually, the designer will feel that the structural aspects (classes and associations) of the design are nearly done. In this scenario, it is assumed that the next phase of the design will address a very different set of issues related to the dynamic behavior of the system. Before making the switch to thinking about these behavior issues, the designer wants to finalize the structural issues as much as possible. As the design nears this milestone, the designer asks, “What is left for me to do?” This question might also arise when the designer is approaching a deadline or even the end of the workday. Alternatively, when the designer realizes that the design is far from reaching the next expected milestone, the designer might ask, “What needs to be fixed or finished?”

Argo/UML’s “to do” list helps answer both of the questions above by displaying all outstanding criticisms in an organized way. Furthermore, Argo/UML can help the designer realize that the design is far from being complete and correct: if the number of outstanding criticisms gets too large, Argo/UML will change the color of the item count above the “to do” list. This provides a meta-cognitive prompt for the designer to switch from constructing new design elements to reflecting on the design and resolving identified problems.

In trying to reduce the number of outstanding items, the designer systematically moves down the list of items and considers each one in turn. Some “to do” items identify errors or incomplete parts of the design. The designer corrects some of these immediately while others require more thought. Some items on the “to do” list contain advice that the designer chooses not to follow in this case or suggested corrections that the designer feels are too risky or difficult. The designer skips past some of these items, leaving them on the list; other items are explicitly declined by pressing the Resolve button on the ToDoItem toolbar. Personal reminders that the designer added previously are now reviewed and some of them are resolved. Over time, the “to do” list is reduced to a few items that are not relevant to the milestone.

Once the explicitly identified criticisms have been resolved, the designer continues to look over the design to try to detect any remaining issues. When considering each design element, the designer looks for common problems based on past experience. Design checklists help augment the designer’s own experience with that of expert designers. One example of this is presented in Scene 4.

After the designer has covered each element of the design document, he or she can gain further confidence in the state of the design by reviewing the design decisions that led up to this point. The history tab contains a time-ordered list of all criticisms raised and how they were resolved. The designer browses the list, giving particular attention to item resolutions that introduced new problems and how those secondary problems were resolved. The designer also uses the design history to check decisions that were not covered by earlier reflection on the design document because they did not result in design

elements. For example, the designer's decision to not include a university administrators class might be documented only in the history. While reviewing past decisions, the designer asks "Why did I do it that way?" Comments entered into design history can help answer that question.

By this point, explicit criticisms have been resolved, checklist items have been considered, and the design history has been reviewed. The designer has also reflected on his or her own decisions and may be better able to apply that experience in the future. Now the designer feels ready to move on to the next set of design issues.

Summary. In this scene, several Argo/UML cognitive support features helped the designer to resolve outstanding design issues, and to feel confident about the structural aspects of the design. Design critics pointed out errors, incompleteness, and other issues that are mechanically detectable. The "to do" list allows opportunistic browsing of outstanding items, but in this scene it was used to systematically browse and resolve those items. Then, design checklists helped prompt and guide reflection on broader design concerns by being at-hand, by supporting systematic review, and by containing items authored by experts. Finally, Argo/UML's design history feature helped the designer review the decisions that lead up to the current state of the design.

5.6 Discussion

The five scenes of the usage scenario presented in this chapter demonstrate how Argo/UML's cognitive support features can help designers in carrying out common design activities. These activities include design construction, error detection, error correction,

reminding, clarifying the intent of the design, answering questions that arise during design, asking the types of questions that expert designers ask, cleaning up the design by resolving outstanding issues, and reviewing design decisions. Two other cognitive support features were not discussed in the scenario: the create multiple feature could have be used during construction (scene 1) and the visual blender feature could have been used to answer questions about creative alternatives (scene 3).

As noted at the beginning of this chapter, each scene is condensed to quickly get to the point of support. In actual design tasks, support opportunities are expected to be more widely spread out over time. However, one key aspect of many of the proposed cognitive support features is that they play an active role in identifying, or even creating, support opportunities. For example, design critics and clarifiers prompt designers to correct errors and create an opportunity for procedural support in error correction. Also, the highlighted number of outstanding items above the “to do” list serves to prompt the designer to switch to reflection on the design, thus identifying the need to systematically address open issues and creating an opportunity to support that need. Another example is the opportunistic search utility which prompts the designer to think of related elements when the designer requests a simple search, thus turning a normal design activity into an opportunity for cognitive support.

CHAPTER 6: Heuristic Evaluation of Cognitive Features

Heuristic evaluation of user interfaces is a practical approach for identifying potential usability problems (Nielsen, 1993). In such an evaluation, a usability expert reviews the design systematically using a checklist of common usability problems and issues. This chapter presents a heuristic evaluation of Argo/UML's cognitive support features.

Method and Goals. Each of the following sections evaluates one proposed cognitive support feature in detail using a variation of the cognitive walkthrough method. A cognitive walkthrough is a theory-based user interface usability evaluation that breaks down an interaction into detailed steps and evaluates each step in terms of the following four criteria (Warton et al., 1994):

- Will the user try to achieve the intended effect?
- Will the user notice that the correct action is available?
- Will the user associate the correct action with the desired effect?
- If the correct action is performed, will the user see that progress is being made toward solving the task?

The programming walkthrough (Bell, Rieman, and Lewis, 1991), a modification of the cognitive walkthrough, considers mental steps performed by the user and refines a model of the knowledge and skills needed by users. This model is called a *doctrine*. An initial doctrine for Argo/UML is shown in Table 6-1. It is assumed that typical Argo/UML users will have the knowledge described in the initial doctrine. This doctrine, along with the cognitive theories described in Chapter 2, is used to justify the usability of each

step in the user interface tasks below. In cases where the initial doctrine is insufficient to justify a step, a new piece of knowledge is added to the doctrine. Each addition to the doctrine is examined again to see if it poses an unreasonable barrier to usage. The final section of this chapter empirically verifies the reasonableness of some additions with a user survey.

Table 6-1: Initial doctrine for Argo/UML

Argo/UML users are familiar with UML and have object-oriented design experience, but they lack complete knowledge of either.
Argo/UML users have used other desktop applications and are comfortable using standard user interface elements such as dialogs, forms, menus, popup menus, progress bars, standard wizards, tabbed panes, text editing fields, tool bars, tree widgets, tool tips, scroll bars, and direct-manipulation.
Argo/UML users are familiar with UML modeling tools and their user interface elements, including diagram editors, table-of-contents widgets, search dialogs, and property sheets.

I have modified the programming walkthrough method to better suit the evaluation of Argo/UML's cognitive support features. First, the evaluation that follows pays special attention to the knowledge needed to perform each step and the knowledge gained in each step. Second, the steps for each user task, or use case, are presented concisely in a table that describes both the required user actions and the tool's reactions. In those tables, minor task variations are handled as alternative steps, while major variations are addressed in separate use cases. Also, optional steps are marked as such. Third, fairly straightforward steps are addressed briefly; the four questions of the cognitive walkthrough method are used implicitly on each step and only identified problems are discussed. Fourth, the level of detail is chosen to fit the specificity of the proposed cognitive support feature. For example, non-modal wizards are a generic approach to providing procedural support for fixing identified design problems; the evaluation of non-

modal wizards will focus on those aspects that differentiate them from standard wizards rather than the specifics of an individual non-modal wizard.

The standard cognitive walkthrough method emphasizes the need for the user to recognize progress on an explicit task. This is needed in part because users often abandon partially completed tasks if they feel that they are on the wrong path. Many of the use cases considered below aid designers in achieving explicit goals. Several of the steps, however, support challenges endemic to the design task without requiring that the designer form an explicit goal before beginning the interaction. Also, many steps are optional because they offer additional support in cases where the normal interaction fails to provide enough to guide the designer. For example, interactions with design critics via clarifiers can help designers to *form* goals related to improving the design, and when simple cues are not enough to prompt the designer to fix identified problems, additional information can be accessed in optional steps. In fact, mixed initative on the part of the designer and the tool is essential to design support systems that try to augment the designer's own decision-making.

6.1 Walkthrough of "To Do" List and Clarifiers

Argo/UML users can access the feedback provided by critics via clarifiers or via the "to do" list. Tables 6-2 and 6-3 show the steps for each of these two use cases.

Table 6-2: Steps for using a clarifier and the "to do" item tab

Step	User Action	System Reaction
A-1	Select a diagram element	Display selection handles and clarifiers
A-2	Form interest in the identified problem	
A-3	Position mouse over clarifier	Display "to do" item headline tool tip
A-4	Read the headline and understand the issue raised	
A-5	Optionally, right-click to access a popup menu	Display pop-up menu
A-6	Optionally, select headline from the popup menu	Show item text in the "to do" item tab
A-7	Read "to do" headline or item text and understand the issue; form the intention and plan to fix the problem	

Table 6-3: Step for using the "to do" list and the "to do" tab

Step	User Action	System Reaction
B-1	Form interest in solving an outstanding design problem	
B-2	Browse headlines in the "to do" list	
B-3	Form interest in a particular criticism	
B-4	Select a headline	Show item text in "to do" item tab
B-5	Read the "to do" item text and understand the issue; form the intention and plan to fix the problem	

Justification of Steps. The first use case begins with an action (A-1) that Argo/UML users perform very often during their work. The tool's reaction to this action includes drawing the standard selection handles found in other direct-manipulation diagram editors, but also includes drawing clarifiers. The wavy-red underline is the most common clarifier in Argo/UML, and it is a familiar indication of trouble to users of other recent desktop applications. This leads the user directly to step A-2. If the user has encountered the identified error before, the clarifier may be enough to prompt the designer to fix the

problem without further interaction with the clarifier. Otherwise, in step A-3, the designer requests additional information via a tool tip. Tool tips are common user interface elements that users often access when presented with a new user interface element. Even if the user does not suspect the availability of a tool tip at first, he or she is likely to accidentally trigger one while working with the diagram and thus learn of its availability. The designer is next expected to read the critique headline tool tip and understand, to some degree, the issue raised. If the designer understands the issue described in the headline well, he or she may jump to step A-7, resulting in a plan to fix the problem without further interaction. Alternatively, the designer can access the details of the criticism by opening a popup menu (step A-5), selecting the feedback item from the "Critiques" sub-menu (step A-6), and reading the "to do" item text displayed in the "to do" item tab. The expected interaction with clarifiers suggests the following additions to the Argo/UML doctrine:

- **Users will realize that there is a popup sub-menu with criticisms that can be selected for additional details.** It is unlikely that users will realize this initially. In fact, Rieman, Franzke, and Redmiles (1995) observe that "users are reluctant to extend their search beyond the readily available menus and controls." The "Critiques" sub-menu will, however, be noticed when the popup menu is activated for other purposes. Since the "Critiques" sub-menu is always present on the popup menu and always contains a list of outstanding criticisms for the selected design element, users should be able to learn of its existence and purpose. I also believe this step to be reasonable because it was inspired by observations of Argo/UML users who saw clarifiers and tried to interact with them. In fact, it was an Argo/UML user who suggested that clarifiers should have popup menus. The reasonableness of this addition has been tested with the user survey described in Section 6.12.

- **Users are able to form an intention and plan to fix the problem identified in "to do" item descriptions.** A study of the VDDE system found that negatively phrased criticism was not effective in prompting designers to fix identified problems. In contrast to VDDE, Argo/UML's feedback is phrased positively and includes specific descriptions of why the criticism should be relevant to the designer's goals and how to

go about resolving the problem. In the end, the effectiveness of feedback text is up to the author of each critic.

The second use case begins with the designer taking the initiative to review and resolve identified problems. Step B-2 requires the designer to look at items in the “to do” pane in the lower-left quadrant of the main Argo/UML window. The designer eventually takes an interest in a particular “to do” item (step B-3) and selects it (B-4). This causes the text of the “to do” item to be displayed in the “to do” item tab. From there, the designer must understand the issue and form a plan to fix it. This interaction with the “to do” list suggests two new doctrine additions:

- **Users will eventually seek to resolve outstanding design problems.** This can be safely assumed based on the cognitive theories of reflection-in-action and opportunistic design.
- **Users know that items in the lower-left pane are “to do” item headlines.** Since it is already assumed that users are familiar with standard user interface elements and desktop applications, it is fair to assume that users will know that items in a tree widget can be safely expanded and selected. Once users explore this widget they will see the “to do” item headlines and full descriptions. These headlines and descriptions serve as examples that demonstrate the purpose of the “to do” list and the “to do” item tab. The majority of users surveyed were able to correctly explain the purpose of the “to do” list.

6.2 Walkthrough of Non-modal Wizards

Table 6-4 shows the use case for non-modal wizards. This use case assumes that the “to do” item text has already been accessed and understood, as described above. This

starting point is needed because wizards are only presented after a “to do” item description has been presented.

Table 6-4: Steps for non-modal wizards

Step	User Action	System Reaction
C-1	Desire support for problem resolution	
C-2	Click “Next>” in the “to do” tab	Display the first wizard panel
C-3	Use widgets in each step of the wizard	Update the design document and wizard progress bar on each step
C-4	Optionally, leave the wizard to work with other features or even other wizards	
C-5	Optionally, return to a partially completed wizard	

Justification of Steps. The first step of using a non-modal wizard is choosing to use it. Then, the designer must click “Next>” to move from the problem description to the first step of the wizard. “Next>” and “<Back” buttons are familiar to anyone who has experience with standard wizards. Indeed, the availability of these two buttons is a strong cue to the user to apply his or her knowledge of standard wizards. In step C-3, the designer interacts with the widgets within a particular wizard, while the tool makes immediate updates to the design. Although the usability of each particular wizard could be evaluated with a separate walkthrough, I am more concerned with the usability of the aspects of the non-modal wizard approach that distinguish it from the standard wizard approach. In particular, the immediate update of the design is unique to non-modal wizards. Immediate updates are part of the direct-manipulation paradigm and are encouraged by the guidelines proposed by Shneiderman (1998) and others.

The last two steps in Table 6-4 are both optional, and it is expected that the designer will work on other tasks between steps C-4 and C-5. In step C-4, the designer may leave

the wizard to use other features of the tool, such as the diagram editor, menu commands, other items in the “to do” list, or even other wizards. At first, designers using Argo/UML may apply their knowledge of standard wizards and assume that they must finish the wizard or cancel it. After a little experience with non-modal wizards, however, designers will encounter a cue card wizard that explicitly directs them to use other tool features. Also, the fact that non-modal wizards do not overlap the main editing window is a strong indicator that other tool features are always available. Designers must desire working with other features and they must feel sure that they will not lose work by leaving a wizard. The cognitive theory of opportunistic design indicates that designers will deviate from prescribed processes, such as the steps of a wizard, when needed information must be looked for elsewhere. Designers who have become familiar with Argo/UML’s “to do” list will know that items are never removed without being resolved. This leads to step C-5, where the designer may return to a partially completed wizard. This is done by selecting the “to do” item via a clarifier or the “to do” list as described above. Designers may return to a partially completed wizard simply by investigating the clarifier or “to do” list as they did originally, or they may explicitly seek out a particular “to do” item that they wish to finish. The walkthrough for the former case is presented above. In the latter case, designers should be able to scan down the “to do” list and identify partially completed wizards when they see blue progress bars. This interaction suggests the need for two additions to the Argo/UML doctrine:

- **Users desire semi-automatic problem correction.** The reasonableness of this assumption depends on the specifics of the problem identified, the difficulty of resolving the problem, and the value added by the wizard. Certainly, designers who have previously perceived wizards as providing high value are likely to apply that perception to non-modal wizards in Argo/UML. The further exploration of this

assumption with a user survey indicated that about half of all users do desire aid in solving identified problems, especially when the problem is difficult or if the wizard makes fixing the problem very easy.

- **Users recognize the small blue bars on "to do" item icons as progress bars.** There is a strong similarity between Argo/UML's "to do" item progress bars and the standard progress bars found in other applications: both are rectangles that extend from left to right as the user makes progress in a multi-step task. The position of Argo/UML's small progress bars, however, is unusual and may prevent users from realizing that they are progress bars. Surprisingly, a clear majority of surveyed users recognized these progress bars.

6.3 Walkthrough of Context Sensitive Checklists

The user interface steps for using context sensitive checklists are much more simple than those needed for accessing criticism via clarifiers or the "to do" list. The mental steps, however, can be more difficult because checklist items can address broader design issues and are less closely linked to the state of the design. Table 6-5 summarizes the steps for using context sensitive checklists.

Table 6-5: Steps for using context sensitive checklists

Step	User Action	System Reaction
D-1	Select a design element	Highlight the element; update the checklist tab
D-2	Form interest in addressing common problems related to the selected element	
D-3	Select "Checklist" tab, if not already shown	Display the checklist for the selected design element
D-4	Read checklist items and consider issues	
D-5	Optionally, request more information	Display a help window
D-6	Optionally, check off the items considered	Display checkmarks

Justification of Steps. The first step requires the designer to select a design element as he or she normally would do. In addition to drawing selection handles, Argo/UML also fills the checklist tab with a checklist appropriate to the selected design element. If the checklist tab is already showing, this change will be seen immediately; otherwise, the

designer will have to select the checklist tab (step D-3). The designer must desire information on common design problems (step D-2) before he or she can be expected to select the checklist tab. This leads to the following addition to the Argo/UML doctrine:

- **Users desire the guidance provided by design checklists.** Designers are expected to need knowledge support because they have limited knowledge and because they may have difficulty in applying the knowledge they possess. The knowledge in checklists, however, may not appeal to designers. The assumption that designers desire checklists has been tested with survey questions. Seven of the twenty users who answered the relevant survey question indicated that they would like to use checklists often, another nine additional users would use checklists occasionally.

In step D-4, the designer must read a checklist item and consider the design issue it raises. Whether this step is reasonable or not depends on the wording of specific checklist items. Two factors that increase the effectiveness of checklist items are (1) the fact that checklist items are context sensitive and contain specific design terms rather than vague pronouns, and (2) the availability of on-line help for many checklist items. In step D-5, the designer may access additional information that explains the issue raised in more detail. In the final step, the designer may check off items as they are considered. This serves as a reminder of progress and prompts the designer to go through the checklist systematically. The option to check off items suggests that the following should be added to the Argo/UML doctrine:

- **Users will check off checklist items as reminders to themselves.** This assumption may be problematic because users typically do not take actions that are not perceived as making progress toward a goal. Checking off items is not required and users may feel no desire to perform unneeded steps. Experienced designers, however, are well aware of the effectiveness of systematic consideration of common problems and the need for reminders during complex design tasks. This doctrine addition is also tested with user survey questions. Almost all surveyed users were able to guess the purpose of the check boxes and half said that they would use them.

6.4 Walkthrough of Design History

Designers can access information about past design decisions using Argo/UML's "History" tab as described in Table 6-6.

Table 6-6: Steps for using design history

Step	User Action	System Reaction
E-1	Desire review of design history	
E-2	Select the "History" tab	Display design history tab
E-3a	Select "Global"	Display list of all history items
E-3b	Select "Selected Element"	Display list of history items related to selected design element
E-4	Read and understand history items	
E-5	Optionally, select a history item	Display details of that item and list the design elements involved
E-6	Optionally, select a design element from the related list	Highlight the item in the diagram pane

Justification of Steps. First, the designer must form the desire to review the history of design choices. The cognitive theory of reflection-in-action indicates that designers do reflect on the state of the design, but it is not obvious that they would seek out design history. Once they have formed the desire to review past design decisions, they must recognize the "History" tab as providing that information (step E-2). In steps E-3a or E-3b, the designer selects global history or chooses to view a subset of the history related to the selected design element. This step seems reasonable because the labels are clear and because the designer can safely experiment with these two settings and understand their effects immediately. Step E-4 is a difficult step that requires the designer to read a one line description of a history item and form an understanding of its meaning. Specifically, designers must recognize history items that relate to current design decisions. Once the designer recognizes a relevant history item, he or she can view its details by selecting it

(step E-5). These details include the full text of the history item and a list of the design elements that were involved in the criticism or modification. The designer may optionally follow a trail of history items by selecting one of the listed design elements to see its history (step E-6).

- **Users will eventually want to review design history**. This assumption is one of the more serious barriers to usage of this feature. Expert designers can be expected to desire design histories, but the majority of users might never desire them. A potential refinement would be to make the history feature more proactively advertised with a "tip of the day" or a visual indication of relevant history in the design diagram itself.

- **Users will find the "History" tab when they want to view design history**. Identifying the proper tab should be easy for users once they have formed the desire to access history. It is likely, however, that they will initially scan the interface when first learning to use the tool and not realize the meaning of the history tab. One possible improvement would be to supply descriptive help when the history tab is first accessed.

- **Users will recognize items in the history tab as past criticisms, changes, and resolutions**. Some of the items in the history list are "to do" items and have headlines that can be recognized by users familiar with the "to do" list. Once users recognize that some of the items are past criticisms, the inference that the others are also historical items should be straightforward. The meaning of the particular types of history items will probably only be discovered after reading the text of a few items.

- **Users will recognize if and how history items relate to current design decisions**. This is the basic assumption underlying the value of this feature. The tool can help users identify which design elements were involved in a given part of the design history and which history items relate to current design elements. But, the user must ultimately make the connection between the historical facts and present design goals. One possible improvement would be knowledge-based aids for history analysis. For example, "history critics" could be defined as a form of critic that identify problematic patterns in the design history, such as one item being constantly revised over time. Design elements that are revised too often are usually design bottlenecks that will require further attention if not removed.

6.5 Walkthrough of Opportunistic Search Utility

Table 6-7 shows the steps needed to use Argo/UML's opportunistic search utility. This feature is an improvement on the standard search utilities found in many commercial

UML tools and other widely used desktop applications. This walkthrough emphasizes the aspects that differ from standard search utilities.

Table 6-7: Steps for using the opportunistic search utility

Step	User Action	System Reaction
F-1	Open the Search window	Display Search window; show help tab
F-2	Enter search criteria	Enable “Search” button
F-3	Press “Search” button	Perform search; display search results in new tab
F-4	Review search results	
F-5	Select search result	Display related design elements in lower table
F-6	Optionally, form an interest in related design elements	
F-7	Review related design elements	

Justification of Steps. Designers commonly use search utilities in programming and design tools as a means of navigation and to get an overview of a certain aspect of the design. In step F-1, the designer opens the search window via a menu command or keystroke. Argo/UML makes finding the search command easy by placing it under the Edit menu, which is the same place that it is found in many other tools. As soon as the search window is displayed, the “Help” tab is displayed in the lower half of the window. The help tab explains the non-standard aspects of the search utility. In step F-2, the designer enters search criteria, such as a wild-card expression for the names of design elements to find or the type of element to find. Most search criteria fields are standard and straightforward, but some may be difficult to understand or use. The emphasis here, however, is on the cognitive support provided when the designer reviews the search results. In step F-3, the designer presses the Search button and Argo/UML creates a new tab with the search results. The new tab is labeled with a concise summary of the search criteria. The designer then proceeds to review the search results (step F-4). Results are

displayed in a table with one result in each row. A designer familiar with standard desktop tools should have no trouble recognizing or understanding the search results, and he or she will naturally click on a search result to learn more about it (step F-5). When the designer selects a search result, the opportunistic search utility also displays related design elements in a second table in the lower half of the results tab. The fact that the items in the lower table are related to the selected result should be clear since they appear after a selection is made. However, the nature of the relationship could be unclear. Step F-6 in Table 6-7 requires the designer to form an interest in a related item. Note that the exact nature of the relationship need not be understood, so long as the designer forms an interest. In fact, it is expected that designers may see coincidental relationships in addition to the rules used by Argo/UML to generate these items. Finally, the designer may review a related result in context by double clicking on it. This step should also be natural to designers who have used other search utilities. This use case suggests the following additions to the Argo/UML doctrine:

- **Users will understand the search window help tab**. Even though the help tab is presented on the first use of the search utility, users may not read or understand it. However, it is reasonable to assume that users will successfully access the help tab if they encounter problems later.

- **Users will associate the label on the result tab with the search criteria**. The two key factors that should make the search result tab labels recognizable are (1) that they include search terms entered by the designer and (2) that they include the asterisk ("*") symbol, which is normally interpreted as a search wild-card character. Only one of the surveyed users failed to understand the meaning of the tab labels.

- **Users will form their own understandings of the relationship between the selected search result and the related results**. The exact meaning of the relationships can be very hard to guess by simply looking at examples. However, users should not need to understand the rules by which related design elements are discovered. Having some obvious relationship rules can help reinforce the fact that the bottom table contains related elements. For example, the attributes, associations, and operations of a class

are obviously related to the class itself. If such obvious rules are omitted, the user is likely to question their initial understanding of the meaning of the related elements table. So long as basic rules are in place, advanced rules should not pose a problem. Surprisingly, eight out of eleven surveyed users who were shown a single screenshot were able to form reasonable understanding of the meaning of the related result items.

6.6 Walkthrough of Opportunistic Table Views

The walkthrough of Argo/UML's opportunistic table views consists of a single use case with several optional steps. The use case is described in Table 6-8.

Table 6-8: Steps for using opportunistic table views

Step	User Action	System Reaction
G-1	Form the desire to use a table view	
G-2	Select the "As Table" tab	Display a table view
G-3	Optionally, change table perspective	Display a table perspective
G-4	Optionally, select highlighting mode	Update highlighting
G-5	Select and edit table cells	Display table values and highlighting
G-6	Optionally, press replay button	Re-display recent highlighting

Justification of Steps. A designer using Argo/UML can edit any semantic aspect of the design via either a diagram view and properties tab or via a tabular view. The diagram view is the default and more familiar of the two. Making the choice to use a tabular design view may require meta-cognitive insight on the part of the designer: i.e., the designer must realize when there is an advantage to using the tabular view.

Once the designer has formed the desire to use a table view, he or she selects the "As Table" tab (step G-2). Choosing the "As Table" tab should be easy because the label is clear, it is centrally located, and users know that they can safely explore tabs. The most commonly useful table perspective is shown by default. Designers may optionally switch to an alternative table perspective from a choice widget (step G-3). Next, the designer

may adjust the highlighting mode to better fit his or her systematic scanning behavior (step G-4). Thinking of one's scanning behavior in advance definitely requires meta-cognitive awareness that the designer may not have. However, even if step G-4 is not carried out, the default row-by-row highlighting mode is the one most likely to be useful. In step G-5, the designer uses the table as normal. This includes selecting and editing table cells to accomplish some editing task. Since it is assumed that the designer specifically chose the table view instead of the diagram view, it is likely that he or she will perform a task that takes advantage of the tabular nature of the table view. For example, the designer is likely to systematically scan the design or make a series of edits to corresponding parts of the design. While the designer is carrying out that task, he or she may need additional information from some other part of the design. The theory of opportunistic design indicates that designers are likely to switch tasks to work with other parts of the design to gather needed information. Upon returning from such a design excursion, the designer may use the "Instant Replay" button to request a replay of the most recent activity in the table view (step G-6). The intent of the replay feature is to help the designer see where he or she left off and thus re-establish a part of the mental task context. However, the effectiveness of the instant replay animation needs to be verified.

The expected interaction with opportunistic table views suggests the following additions to the Argo/UML doctrine:

- **Users will understand the advantages of a tabular view and choose it over a diagram when appropriate**. This is one of the biggest assumptions of the opportunistic table views feature. As discussed below, a survey of users indicated that they did not form a desire to use tables when systematically checking a design diagram. It seems likely that users could learn the value of tables with experience, but encouragement will be needed for most users to recognize that value. Potential

refinements to address this barrier to usage include “tip of the day” dialogs or coaches that watch the user’s actions and suggest more effective strategies.

- **Users will notice the “As Table” is tab**. Noticing the “As Table” tab should be easy for users because it is shown in an area of the main window where users access many different Argo/UML features. All but one of the surveyed users were able to access the table view.

- **Users will notice the choice of alternative table perspectives in the table tab**. Finding the table perspective choice is not difficult because it is located above the table, near the column headers. In many desktop applications, table views are customized via the column headers, so that is a natural place to look. All of the surveyed users who responded to the relevant question gave answers that indicated that they would be able to change the table perspective.

- **Users will plan systematic scans of the design**. Expert designers certainly do perform systematic scans of the design and make systematic changes. However, teaching designers this expert behavior will require more support than is currently provided by this feature. A potential refinement would be a task-tool-matching dialog or help file. That service would allow the user to select a task-level goal from a list of supported tasks and then advise the user on which perspectives to scan during that task.

- **Users will recognize the purpose of the “Instant Replay” button**. The majority of users surveyed did not understand the meaning of this button label. A possible refinement would be to relabel this button “Show recent highlights”. However, the majority of surveyed users thought that pressing the “Instant Replay” button was a safe operation, and once they saw the instant replay animation, they understood it’s meaning.

- **Users will see where they left off based on the instant replay animation**. Users can be expected to understand the instant replay animation in the context of use because it simply replays actions that the designer recently performed. There was some confusion among surveyed users as to whether the final row highlighted was the row that they should continue to work on or if that row had already been considered. Allowing designers to be off-by-one is acceptable if it results in a row or column being considered twice, but it is not acceptable if it results in a row or column being skipped. For that reason, the animation should stop short and not include the most recent row selected.

6.7 Walkthrough of Navigational Perspectives

Designers use Argo/UML's navigational perspectives by performing the steps described in Table 6-9.

Table 6-9: Steps for using navigational perspectives

Step	User Action	System Reaction
H-1	Form question about a design structure	
H-2	Choose a perspective to help answer question	Update Navigation pane
H-3	Expand tree widget in navigation pane	Show tree according to chosen perspective
H-4	Understand tree and use it to answer questions	

Justification of Steps. As with several of Argo/UML's cognitive support features, the designer must first realize that he or she needs a certain type of information. In the case of navigational perspectives, the designer must form a question about a design structure (step H-1). The theory of reflection-in-action and other cognitive theories suggest that raising and answering questions about the design is a key part of the design process. In step H-2, the designer must choose a navigational perspective from the choice widget above the navigation tree. To accomplish this step, the designer must first recognize that the choice widget is the proper affordance. Then, the designer must choose a perspective that will help answer the question raised based on the name of the perspective. Some of the perspective names are very clear because they use standard terms like "inheritance," while others may not be so clear. If this is the first use of the selected perspective, it will be completely collapsed and must be expanded (step H-3). Expanding the tree should be natural because, at this point, the tree has obviously changed, and expanding is the normal operation on a collapsed tree. In step H-4, the designer must understand the design structure that is shown. As with the opportunistic search utility, there is the possibility that

the designer will see a different structure than the one intended. This could be useful, or it could be misleading. This use case has identified the following doctrine additions:

- **Users will form questions about design structures that the navigational perspectives can help answer**. Many of the predefined design perspectives are specifically designed to address commonly occurring design questions by making key design structures clearly visible. For example, the inheritance and reachability perspectives are useful in answering many common questions.

- **Users will recognize the perspective choice widget**. The position of the perspective choice widget directly above the navigation tree should make it easy for users to find and recognize this widget. One potential refinement would be to label the perspective choice widget with “Perspective:”, as is done in the ObjectDomain tool.

- **Users will choose an appropriate perspective based on its name**. As noted above, some of the perspectives have very clear names than use standard terms for key design structures. Once the user uses one perspective successfully, they can be expected to explore the others. However, if the user does not understand a given perspective, he or she is unlikely to use it when needed. In a survey question related to systematic scanning of all associations in the design, two out of eight surveyed users indicated that they would attempt to use the “association-centric” perspective. Argo/UML, does not provide a predefined “association-centric” perspective, so the users must have understood the naming conventions used by other perspectives to invent a new name fitting the same convention.

- **Users will understand the design structure being shown in the navigation pane**. Some perspectives are clear, others are not. Understanding a perspective that is not clear based on its name alone will require a designer to understand the design structure being shown in the tree widget. This could be a problem, especially if the designer forms an incorrect understanding of the tree structure. It is possible that designers will look at the navigational perspective configuration window to gain an understanding of a perspective, but that is unlikely. One refinement that would help overcome this barrier to usage would be to attach a descriptive comment to each navigational perspective. In cases where a perspective has no comment, a default description could be generated by combining short descriptions of the rules that make up the perspective.

6.8 Walkthrough of Broom Alignment Tool

Table 6-10 shows the steps needed to use Argo/UML's broom alignment tool. This walkthrough is specified at a lower level of detail than most of the others because the broom uses a novel interaction that does not make use of standard widgets.

Table 6-10: Steps for using the broom alignment tool

Step	User Action	System Reaction
I-1	Mentally visualize diagram elements in semantic groups	
I-2	Form desire to express semantic groups as visual groups	
I-3	Drag diagram elements roughly into desired positions	
I-4a	Enter broom mode via broom toolbar icon	
I-4b	Enter broom mode via control-click in diagram area	
I-5	Drag the mouse in the diagram area	Draw broom
I-6	Push diagram elements into alignment	Move diagram elements
I-7	Optionally, drag backwards to undo unwanted movements	Move diagram elements back toward their original position
I-8	Optionally, press spacebar	Evenly space the elements on the broom
I-9	Release the mouse button	Exit broom mode, go to selection mode

Justification of Steps. The cognitive theory of secondary notation indicates that designers will desire diagrams that make effective use of visual properties such as alignment and spacing. Based on this theory, steps I-1 and I-2 should occur naturally. Since it is assumed that designers are experienced with diagramming tools, step I-3 should not be a problem. In fact, designers are likely to move diagram elements into rough alignment as a way to quickly evaluate the visual impact of a given layout. In step I-4, the designer can use the toolbar button for the broom mode (step I-4a) or use control-drag (step I-4b). The toolbar button provides a visual affordance to start broom mode, but the designer may have difficulty recognizing the broom mode icon. The control-drag option is less likely to be discovered, but it might be learned from the documentation.

Once the designer has changed modes in the diagram editor, he or she is likely to click or drag in the diagram area (step I-5). Furthermore, a message in the status bar of the main Argo/UML window briefly explains the broom and prompts the designer to drag. The status bar message is, “Push objects around. Return toggles pulling. Spacebar distributes.” If the designer merely clicks, a blue plus-sign is shown until the user eventually drags the mouse. Dragging produces an immediate visual effect that indicates that something is happening. The shape of the broom tool provides a pushing affordance that should lead designers to step I-6. Even if the shape of the broom does not immediately suggest its behavior, the designer should naturally discover and understand the broom’s push-to-align action. In step I-7, the designer may optionally undo the movement of diagram elements by moving the broom in the opposite direction as the original drag. Again, the existence of this feature is not obvious, but it is likely to be invoked accidentally, and once it has been seen its usefulness will be obvious. Alignment by itself does not completely achieve the goal of forming visual groups; even spacing is also needed. Designers may evenly space the diagram elements on the broom by pressing the spacebar (step I-8). The availability of this action is not obvious, but it is mentioned in the status bar whenever the broom is in use. Once the designer has pressed the space bar, its effect is immediately visible, and the message “space evenly” is shown behind the broom. The final step (I-9) is simply to release the mouse button.

- **Users will desire alignment and even spacing in UML diagrams**. When asked to select the most readable of three diagrams, the majority of surveyed users selected one that used alignment as an effective secondary notation. Even if designers are not aware of their desire for secondary notation, they very often spend time trying to get their diagrams to look neat and orderly by aligning diagram elements.

- **Users will move diagram elements into rough position while deciding on layout**. When asked how they would improve a poor layout to make it more clear, the majority of users indicated that they would move nodes into alignment. Also, the laboratory study of alignment tools showed that most subjects moved nodes into rough alignment. Users who are familiar with standard alignment tools often move nodes into rough alignment so that the effect of standard alignment command, such as "align tops," will be more predictable.

- **Users will associate the broom mode icon with the broom**. This assumption is very unlikely to be met by first time users since the broom is not related to any commonly used desktop application feature. In a survey of users, very few knew the meaning of the broom icon, and the tool tip "Broom" did not provide much assistance. However, once the functionality of the broom has been discovered and understood, the icon can become familiar because it shows the recognizable shape of the broom. One refinement to this feature would be a more descriptive tool tip such as, "Broom alignment tool."

- **Users will feel safe in experimenting with broom mode**. The surveyed users indicated that they thought that the broom icon was a safe button to press. This is implied because of the location of the broom icon next to the selection mode icon on the toolbar. Once users are in broom mode and move some objects around, they will realize that the broom does affect the state of the design, but it does so in a safe and understandable way.

- **Users will discover that control-drag also starts broom mode**. It is very unlikely that users will discover this without being told explicitly. One refinement would be to add a tool tip, "tip of the day," or status bar tip to explain that the broom can also be accessed via control-drag.

- **Users will guess that the broom can push objects based on the shape of the broom**. Based on the survey results, a majority of users will guess the functionality of the broom when they first see it. Furthermore, in actual usage the fact that the broom is attached to the mouse pointer should lead to immediate experimentation, so it is reasonable to assume that even those who do not understand it immediately will understand it after using it once. The shape of the broom also indicates the direction of alignment and the scope of its interaction with the diagram elements.

- **Users will understand the push-into-alignment metaphor**. The majority of users surveyed understood the functionality of the broom after seeing two screenshots of its use. It seems very reasonable to expect users to understand the push-into-alignment metaphor.

- **Users will understand that backing up allows objects to return to their initial position**. Based on four screenshots of the broom reversing direction, surveyed users were able to understand this aspect of using the broom very clearly. In actual usage,

the rate of understanding is expected to be even higher since experimentation and visual feedback will be immediate.

- **Users will realize that pressing the spacebar will distribute objects**. It is unlikely that the user will discover this feature without being told. Argo/UML already includes a status bar message telling the user to press the spacebar, however status bar messages are often ignored, especially when the user is involved in a direct manipulation. It is possible, however, that the user would eventually notice the message and try to use the spacebar. One possible refinement would be to provide help on using the broom when it is first used, or to simply leave a status bar message visible after the broom mode has been exited, if spacing was not performed.

6.9 Walkthrough of Model-based Layout

Expected usage of Argo/UML's proposed model-based layout feature consists of the two use cases described in Table 6-11 and Table 6-12.

Table 6-11: Steps for using grid-based layout

Step	User Action	System Reaction
J-1	Form desire to automatically redo diagram layout	
J-2	Issue "Layout Diagram..." command	Open "Model-based Layout" window
J-3	Select "Grid-Based" tab	Show tab, including attribute fields, preview pane, and "Layout Diagram" button
J-4	Choose attributes to be used in layout	Update preview pane
J-5	Divide attribute value ranges	Update preview pane
J-6	Optionally, order attribute value ranges on axes	Update preview pane
J-7	Press "Layout Diagram" button	Layout diagram using grid constraints and simple geometric rules

Table 6-12: Steps for using region-based layout

Step	User Action	System Reaction
K-1	Form desire to automatically redo layout	
K-2	Issue "Layout Diagram..." command	Open "Model-based Layout" window
K-3	Select "Region-Based" tab	Show tab, including constraint field, region drawing pane, and "Layout Diagram" button
K-4	Draw regions to be used in layout	Update region drawing pane
K-5	Assign constraints to regions	Update region drawing pane
K-6	Press "Layout Diagram" button	Layout diagram using constrained regions and simple geometric rules

Justification of Steps. Both use cases begin with the desire to automatically redo the layout of a diagram. This desire occurs naturally during design and is commonly found and used in other CASE tools. Likewise, the second step is also reasonable. The final step in each use case is also very straightforward. Even if all the other steps are skipped, the designer can simply press the "Layout Diagram" button to produce a layout that is as good as that produced by other CASE tools. Even if a designer does not initially desire model-based layout, it is likely that they will discover this cognitive support feature eventually through normal usage.

In step J-3, the designer must select the "Grid-Based" tab. Accessing a clearly visible tab should not be a problem because exploring tabs is always a safe operation. Next, the user must choose attributes to control the layout. Since this aspect of Argo/UML's model-based layout is not found in other tools, designers may not understand what is being requested. Furthermore, designers may have difficulty choosing the particular model element attributes needed to achieve the desired layout. Once the attributes are selected, value ranges must be defined. For some attributes, this can be trivial or automatic; for others, determining appropriate value ranges may require trial-and-error. The layout preview pane quickly shows a rough approximation of the layout to help designers explore alternatives. In step J-6, the designer may reorder the rows or columns or move a given attribute from a row to a column or from a column to a row. Reordering should be easy to accomplish via the up and down buttons to the right of the attribute list. The first use case suggests the following addition to the Argo/UML doctrine:

- **Users will be able to select model element attributes to control model-based layout**. Model-based layout uses the attributes of model elements, such as name,

visibility, isAbstract, or tagged values. Before the user can pick an attribute to use, he or she must form a question about the design that can be answered by seeing the diagram laid out in a particular way. Designers may have difficulty bridging the gap from questions to layout. One potential refinement to this feature that could reduce this gap would be to offer predefined model-based layouts that help answer common design questions. Another refinement would be to document each predefined layout with a description of its purpose.

In step K-3, the designer selects the "Region-Based" tab. Then, in steps K-4 and K-5, the designer must define geometric regions for the layout and assign constraints to them. It is expected that steps K-4 and K-5 will be done repeatedly as the designer refines the layout. The following additions to the Argo/UML doctrine are needed for the designer to successfully define and refine the region-based layout:

- **Users will understand the region-based layout concept**. This assumption relies on the users' previous experience with diagram layout in other contexts. Users who have prepared technical charts and diagrams in the past are likely to have encountered this concept. Other users may also be able to use this feature after they have learned the concept from the help tab or from experimenting with the Grid-Based layout specification tab.

- **Users will be able to specify constraints for regions**. Software designers are very familiar with boolean conditions, so the actual constraint expression itself should not be a problem. A bigger problem is the user's need to bridge the gap between visualization goals and constraint choices. One refinement that could help address this problem would be to provide predefined constraints with descriptive comments.

- **Users will understand the interactions between regions**. The rules for interactions between regions may seem complex or unexpected to some users. Users should be able to learn these rules by directly manipulating the regions in the preview pane and seeing the approximate layout immediately. The interaction rules should also be clearly explained in the help tab.

6.10 Walkthrough of Selection-Action Buttons

The walkthrough of Argo/UML's selection-action buttons consists of one use case with three alternative conclusions.

Table 6-13: Steps for using selection-action buttons

Step	User Action	System Reaction
L-1	Form desire to expand an existing node	
L-2	Select a diagram node	Show handles and appropriate SABs
L-3a	Click a selection-action button	Create node and edge in the default position
L-3b	Drag from a SAB to empty space	Create node and edge in the position indicated
L-3c	Drag from a SAB to a target node	Create edge between original and target nodes

Justification of Steps. Designers naturally form the desire to add new nodes and edges to the diagram as part of the diagram construction process. New diagram elements may be the initial elements in new clusters of related elements, or they may elaborate on existing clusters of classes or states. The standard toolbar interface must be used for initial diagram elements. However, if the designer wants to elaborate on existing diagram elements by adding new related elements (step L-1), he or she can use the selection-action buttons.

Selection-action buttons are implemented as an enhancement to normal diagram element selection. As such, the designer will encounter them during the normal course of selecting diagram elements (step L-2). Steps L-1 and L-2 may be reversed if the selection-action buttons serve as a prompt for the designer to consider adding new elements.

The designer can choose three alternatives for the final step. In alternative L-3a, the designer simply clicks the selection-action as he or she would click a toolbar button. This

has the immediate effect of producing a new diagram node and edge of the appropriate type in a default position near the originally selected node. In alternative L-3b, the designer drags from the selection-action button to the desired position of the new node. This alternative is not immediately obvious because buttons are usually not dragged. However, once the designer is familiar with alternative L-3a, he or she may be dissatisfied with the default position and seek to specify the position via dragging. Alternative L-3b might also be discovered accidentally if the designer tries to click and quickly move the mouse to select the new node. While the designer is dragging, a rubberband line is drawn to give immediate feedback and the status bar shows the message, “Drag to define an edge (and a new node).” The final alternative, L-3c, requires the designer to drag from a selection-action button to an existing node. This results in a new edge between the two nodes. Designers may attempt the alternative interaction simply because it seems to be an obvious way to accomplish this goal. The status bar message also suggests that it is possible to define a new edge without defining a new node. Furthermore, the rubberband line feedback mechanism is the same one used when adding an edge between nodes in many diagramming tools, so it may prompt the designer to assume that the same functionality is available in this context.

- **Users will recognize the function of each selection-action button as a variant of the standard toolbar buttons**. In fact, it is likely that the meaning of the standard icons is clearer when they are used as selection-action buttons because their position suggests their function. For example, the generalization tool has an icon consisting of a vertical line with a hollow, triangular arrowhead. This shape is the standard shape used in UML, yet it is not very easily distinguished from other types of arrowheads used in UML. When the same icon is shown at the top of a class node, its meaning should be clearer because its position suggests the normal direction of generalization edges, and because only the most commonly used options are offered to the user.

- **Users will discover the option to drag the selection-action buttons**. Users are expected to discover the option to drag because the normal tools for creating edges involve dragging. Surveyed users did not indicate that they would immediately discover this aspect of the selection-action buttons. Instead, most users would always use the single-click aspect and then position the newly created class.

- **Users will consider dragging from node to node to make a new edge without making a new node**. Once users have dragged from selection-action buttons, the option to drag to an existing node seems fairly natural. All but one of the subjects in the selection-action button laboratory study seemed to have no problem with this. Likewise, only two out of fourteen surveyed users had a problem with this aspect of the selection-action buttons.

6.11 Walkthrough of Create Multiple

Table 6-14 and Table 6-15 show the steps needed to use Argo/UML's create multiple feature.

Table 6-14: Steps for creating multiple design elements by pattern

Step	User Action	System Reaction
M-1	Form desire add several related design elements quickly	
M-2	Issue "Create Multiple..." command	Open "Create Multiple" window
M-3	Select "By Name" tab	Show a list of available pattern names, description and parameter panes, summary pane, and "Create" button
M-4	Select pattern by name	Display pattern description and parameters
M-5	Optionally, request more information on pattern	Open help window
M-6	Supply element names and any other pattern parameters	Update summary pane
M-7	Press "Create" button	Add specified design elements to current diagram

Table 6-15: Steps for creating multiple design elements by form

Step	User Action	System Reaction
N-1	Form desire add several related design elements quickly	
N-2	Issue “Create Multiple...” command	Open “Create Multiple” window
N-3	Select “By Form” tab	Show a list of available forms by name, form pane, summary pane, and “Create” button
N-4	Select form by name	Display selected form with data entry fields and pattern explanation links
N-5	Optionally, click on pattern name link	Open help window
N-6	Supply element names and any other form parameters	Update summary pane and color elements black
N-7	Press “Create” button	Add specified design elements to current diagram

Justification of Steps. As with the model-based layout use cases, both of the use cases for create multiple begin with forming a desire, followed by opening a secondary window, and end with pressing a button to confirm the operation. These three steps should not be a problem because they are familiar to users who have experience with other desktop applications.

In step M-3, the designer must select the “By Name” tab. This step seems reasonable because the designer cannot do anything in the help tab, and because exploring tabs is always a safe operation. Next, the user must select a pattern by name from a list. This could be a difficult step because the designer may not recognize the meanings of the pattern names. Even if designers do not initially understand every pattern, they can be expected to explore some of the patterns in the list because list selection is always a safe operation. In step M-5, the designer may optionally request more information about the selected pattern. This step should not be difficult because the “More Info” button is clearly visible. In step M-6, the designer supplies names for new design elements to be created and the names of existing elements onto which the new elements will be grafted.

The reasonableness of this step depends on the specific prompts provided by each design pattern configuration panel.

Step N-3 in the second use case requires the designer to select the “By Form” tab. Again, this should not be a problem. Next, the designer must select a form by name from a list. It is not expected that the designer will be able to choose the correct form on his or her initial attempt, instead the designer is expected to explore various forms until he or she sees one that looks relevant. A designer must gauge the relevance of a particular form to his or her immediate needs based on the brief description of the form and the design structures that are visually evident in the form. Once a form is selected, the designer may optionally click on one of the design pattern links to access more information on the intent and applicability of that design pattern. Clicking on an underlined link should not be a problem for users that are familiar with web browsers. In step N-6, the designer fills in the form with the names of new design elements and the names of existing elements to graft the new elements onto. The process of entering the names should not be a problem because it only involves using standard text entry widgets. However, choosing which widgets to fill in and which to leave blank requires the designer to understand the grafting rules. These two use cases suggest the following doctrine additions:

- **Users will recognize pattern names**. Much of the object oriented design community has studied design patterns, so some users will definitely be able to recognize and pick design patterns by name. In fact, the Together/J design patterns feature relies on users having this knowledge. Other users may not recognize pattern names and will need to explore the items on the list. The quality of the descriptions and the labels of the pattern parameters is key to this exploration process. Users who have difficulty with the “by pattern” use case may find the “by form” use case easier.

- **Users will understand grafting rules**. The grafting rules are fairly simple, they are explained in the help tab of the create multiple window, and their results are

immediately previewed in the summary pane. Together, these three aspects of the create multiple feature should make it possible for users to understand the grafting rules.

- **Users will recognize forms that fit their mental image of the design structure they wish to create**. Designers often visualize the result they desire before they begin constructing it. The visual presentation of design patterns in the create multiple window should make it easy for designers to match their mental images. Geometric layout differences between the designer's mental image and the offered form might be addressed by refining this feature to include an option to flip the form horizontally or vertically. Also, the forms offered by the create multiple feature can also help shape the designer's desire to create related design elements. Designers are likely to become familiar with the forms offered by the create multiple feature and use it when they recall that a given form is appropriate for their current need.

6.12 Discussion and Validation

Performing the cognitive walkthroughs of the proposed features has been a productive step in my feature generation method. Breaking down the user's expected interaction with each feature has forced me to think through all the details and has highlighted several potential problems and refinements. One of the main dangers of user interface design is assuming that the user knows everything that the user interface designer knows. Several of the elements of the Argo/UML doctrine may be barriers to usage. I have conducted a set of user surveys to probe the knowledge of Argo/UML users and determine which elements of the doctrine are, in fact, problematic. Removing these problems will require refining the cognitive support features to eliminate the need for certain knowledge or skills. The surveys and refinements are discussed below.

User surveys. One straightforward way to determine if Argo/UML users possess a given piece of knowledge is simply to ask them questions that require that knowledge. In August 1999, I conducted an anonymous survey of registered Argo/UML users. The survey consisted of three questionnaires with ten to twelve questions each. Most of the

Imagine that you are using Argo/UML. You select a class in the class diagram editor. When the class is selected you see a wavy red underline under the name of the class. What do you think the wavy red underline indicates?

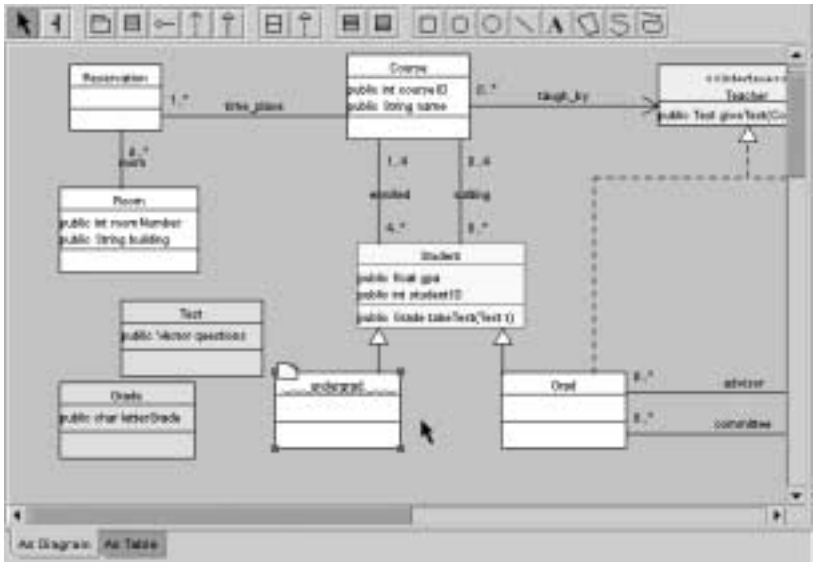

Figure 6-1. A survey question on clarifiers

questions presented a screenshot and asked the user for his or her thoughts on the purpose of each particular widget. An example question is shown in Figure 6-1. Some questions used three parts to test the degree of knowledge support provided by a feature: the first part asked how the designer would address a given problem, the second part asked the designer to explain the purpose of particular feature, and the third part described the support provided by the feature and asked again how the designer would address the problem in light of that support.

For each questionnaire, I constructed a set of web pages with one question per page. Each page included a text field or multiple choice widget and a “Submit” button. Subjects were instructed give their first thoughts if they did not know the answer, and they were told not to return to previously answered questions. For each questionnaire, an email

message with the web page address was sent to a subset of registered Argo/UML users. Approximately one hundred and fifty messages were sent for each questionnaire, and they were sent to users in alphabetical order. This method of validating the cognitive walkthroughs yielded very rapid results: for each questionnaire, I received more than a dozen responses within the first two days.

Survey results. The results of particular questions are described above. Tables 6-16 through 6-18 summarize the survey results. The first questionnaire probed the issues raised by the walkthroughs of clarifiers, the "to do" list, non-modal wizards, and checklists. The second questionnaire probed the issues raised by the walkthroughs of Argo/UML's opportunistic search utility and table views. The third questionnaire probed the issues raised by the walkthroughs of the broom alignment tool and selection-action buttons.

Table 6-16: Questionnaire results for clarifiers, "to do" list, wizards, and checklists

Question	Success	Partial	Failure
Recognize meaning of clarifier	15	0	6
Able to access clarifier tool tip	13	2	5
Tool tip enough information to fix	18	0	3
Able to access "to do" item via pop up menu	13	1	5
Meaning of "to do" list	10	5	4
Desire to fix problem identified	19	0	1
Description had enough information to fix	19	0	1
Desire for wizard to fix problem	9 wizard, 11 manual		
Desire for future wizards	13 wizard, 7 manual		
Understand how to use wizard	17	1	1
Recognized non-modal wizard progress bar	13	1	5
Understood "to do" item count	0	19	1
Expected frequency of attention to feedback	14 episodic, 3 immediate, 3 end of project		
Desire to use checklists	4 unlikely, 9 a few, 7 a lot		
Understood purpose of checkmarks	18	1	1
Desire to check off checklist items	10 often, 6 rarely, 4 never		

Table 6-16 shows that the majority of the assumptions added to the Argo/UML doctrine are reasonable or marginal. One possible problem is that a fair fraction of users confused the “to do” list with the checklist tab. Also, the count of items at the top of the “to do” list was clear, but the use of a plus or minus sign to indicate the trend of the size of the “to do” list was not clear to anyone. The non-modal wizard progress bar was recognized much more easily than I would have expected.

Table 6-17: Questionnaire results for opportunistic search and table views

Question	Success	Partial	Failure
Start search	10	1	1
Expected search results	12	0	0
Understood search tabs	9	1	1
Understood related results	8	0	3
Use related results	8 yes, 3 no		
Systematic strategy	4 search utility, 2 nav perspective, 1 table, 1 visual		
Access table view	9	1	1
Understand table	11	0	0
Find table perspectives	10	0	0
Scanning behavior	2 visual scan, 9 select each row		
Recognize instant replay button	1	4	4
Instant replay safety	4 dangerous, 7 safe		
Understand instant replay	5	3	2
Resume from excursion	7	2	1

Table 6-17 shows the results of the questionnaire on the opportunistic search utility and opportunistic table views. The search utility seemed understandable to the majority of users. However, a fair number of users had difficulty with the related results table. When asked how they would systematically scan all the associations in the design, the most common answers were to use the search utility or navigational perspectives. This indicates that many designers will not realize that tables are an effective user interface for

systematic design review. However, it may have been biased by the fact that the proceeding questions all dealt with the search utility. When the surveyed users were asked how they would access and use the table views, the majority gave answers indicating that they would be able to use these features. The “Instant Replay” button was not very recognizable, but it was assumed to be safe, and about half of the surveyed users understood its purpose once its animation was seen. The animation question probably would have been answered more positively if an actual animation had been shown in the question rather than a sequence of four small screenshots. In general, the survey results are pessimistic because they ask questions about usage without allowing the users to actually work with the tool.

Table 6-18: Questionnaire results for broom and selection-action buttons

Question	Success	Partial	Failure
Diagram style choice	13 aligned, 5 unaligned		
Recognize broom icon	2	5	7
Recognize broom safety	6 dangerous, 10 safe		
Recognize “broom” tool tip	2	4	9
Predict broom action	9	1	4
Understand broom action	14	0	1
Understand broom reverse	12	0	2
Recognize selection-action buttons	11	1	2
Predict selection-action button action	11	2	1
Guess drag option	1	12	1
Connect existing classes	7	5	2

Table 6-18 shows the results of the questions on using the broom and the selection-action buttons. First, the surveyed users shows a marked preference for diagrams that used alignment as a secondary notation as opposed to misaligned diagrams. Few users could recognize the broom icon, and the tool tip “broom” did not provide any help.

However, once users saw the broom in the diagram pane, almost two-thirds were able to predict its behavior. Once they were shown screenshots of the broom in action, all but one user recognized its usefulness. A second user had trouble understanding that reversing the broom direction served as a kind of undo, otherwise it was clear to all users. Most users recognized the selection-action buttons as variants of the toolbar buttons and could predict that pressing a selection-action button would create new design elements that were related to the current selection. When asked how they would create a new class at a desired position on the diagram, most users said that they would first use a selection-action button to create the class in its default position and then move the class to the desired position. More users were able to guess the possibility of creating a new edge by dragging from the selection-action button of one class to another existing node. However, several users said that they would prefer to use the standard tool bar buttons to create edges between existing classes. Again, the survey is pessimistic in its evaluation because users are asked to explain what they would do rather than actually use the tool. For example, if users were actually using Argo/UML, the difficulty of using the standard tool bar buttons would encourage them to use the selection-action buttons.

CHAPTER 7: Empirical Evaluation of Cognitive Features

This chapter presents empirical studies of Argo/UML usage. The following five sections describe three laboratory usability studies, observations of classroom usage, and a summary of feedback from internet users.

My feature generation approach produces fairly independent features. I have demonstrated these features in the context of the Argo/UML tool, but they can be applied to other design domains and tools. To evaluate individual features, I have used controlled laboratory studies that focus on specific design tasks. To evaluate the overall tool, I have gathered and analyzed feedback from actual users.

7.1 Pilot User Study

Goals. The goals for the pilot user study were to practice the skills needed to conduct user studies and to resolve any serious usability problems that might interfere with further studies.

Setting. In June 1998, I conducted a pilot laboratory study on one of the first released versions of Argo/UML. This study was approved by the UCI human subjects research committee as study HS98*224. Two subjects were asked to use Argo/UML to complete the task shown in Figure 7-1. Subjects were given brief demonstrations and instruction on using Argo/UML and then proceeded to use the tool as best they could. Subjects were carefully observed and the problems they encountered were noted. The tool was updated

to address problems identified with the first subject before the second subject was observed.

Results. This study resulted in progress on both of its goals: my understanding of tool evaluation improved, and the tool itself was improved to help clear the way for further evaluation.

The biggest change in my evaluation plans resulted from the way subjects dealt with the problem statement. The problem statement given was meant to be somewhat complex

> You are a professional software designer working at a company that has recently been given a contract to design and implement a new registration system for universities like UCI. Your task is to read the requirements given below and come up with a design.
>
> TELE has been such a success that the company that developed TELE has been bought by Microsoft and will now develop similar systems for sale to many universities. TELE-2 is substantially similar to TELE, but will not handle fee payment or grades, and TELE-2 will keep track of teaching assistants. TELE-2 should be flexible and extensible enough to be customized to the needs of different universities.
>
> The purpose of the TELE-2 system is to keep track of information about (1) courses offered at the university, (2) the relationship between courses, and (3) the administrators, faculty, staff, and students involved in each course. TELE-2 will have a menu-based user interface that can be accessed via telephone or the web, but you will not work on that part of the project.
>
> Courses are taught by an instructor and attended by six or more students. Courses can be lectures, seminars, tutorials, or independent study. Courses may have associated discussion sections, lab sessions, or studio sessions. Courses may have restrictions on the students who are able to enroll. For example, some courses are only open to undergraduates who major in that subject, and some courses are only open to honors students.
>
> Professors or lecturers teach courses. Graduate students teach the sessions associated with courses. Lab sessions must also have a Professor who is responsible for the safety of the lab. Courses and their associated sessions are held in a room. Each room has a seating capacity. Students may enroll in courses if there is available seating; otherwise they are placed on a waiting list.

Figure 7-1. Task for pilot study

and open-ended so as to give subjects a reason to make design decisions. However, it also provided enough concrete facts that subjects felt they needed to enter those facts into their designs before proceeding with any creative design work. Since there were many facts to enter, the entire testing session was taken up by transcription rather than design decisions. In fact, subjects used the problem description sheet to check off design elements as they entered them. As a result, later laboratory studies focused on smaller tasks involving individual design features rather than large design tasks involving the whole tool.

The second change in evaluation plans resulted from the failure of subjects to effectively follow the think-aloud protocol. Initially, I had hoped that subjects would report their own thought processes and that I could use that data to measure the perceived complexity of their task and identify any difficulties. As it turned out, subjects lapsed into silence whenever they encountered difficulty.

The third major realization about laboratory user testing was that Argo/UML simply was not ready to be evaluated as a complete tool. At that time, too many missing features, basic usability problems, and outright defects made testing for subtle advantages impossible. All of the problems detected in the pilot study have now been addressed, but the emphasis remains on evaluating individual features in laboratory studies and evaluating the overall tool via interactions with actual users.

Two major improvements to Argo/UML resulted from the pilot user study. First, the need for direct text editing in the diagram pane was seen to be of key importance. At the time of the pilot study, direct text editing was not supported: subjects needed to select design elements from the navigator pane and edit their properties in the “Properties” tab.

Now, Argo/UML users may edit names, attributes, operations, and state transitions directly in the diagram pane. Second, the need for clarifiers became obvious because subjects worked through an hour of design construction without ever switching mental modes to reflect on the design. The theory of reflection-in-action indicates that designers will periodically switch between reflection and construction, but there is no reason to believe that they will do so unaided within a one hour laboratory session. Clarifiers appear directly in the diagram editing pane and visually prompt designers to consider feedback from critics during design construction.

7.2 Broom User Study

Goals. The goal of this second laboratory study was to evaluate the ergonomic and cognitive impact of the broom alignment tool. This study also served to test the usefulness of a technique for measuring short-term memory load.

Setting. In January 1999, Michael Kantor and I conducted a laboratory study that compared the broom alignment tool with standard alignment commands. This study was approved by the UCI human subjects research committee as study HS98*552. We later published a conference paper that reported the results of this study (Robbins, Kantor, and Redmiles, 1999).

In this study, subjects were asked to position diagram nodes into visual groups to reflect various semantic groupings. Nodes were initially placed near the top of the diagram in no particular order or grouping. For each diagramming task, subjects worked once with the broom and once with the standard alignment tools, in random order. This

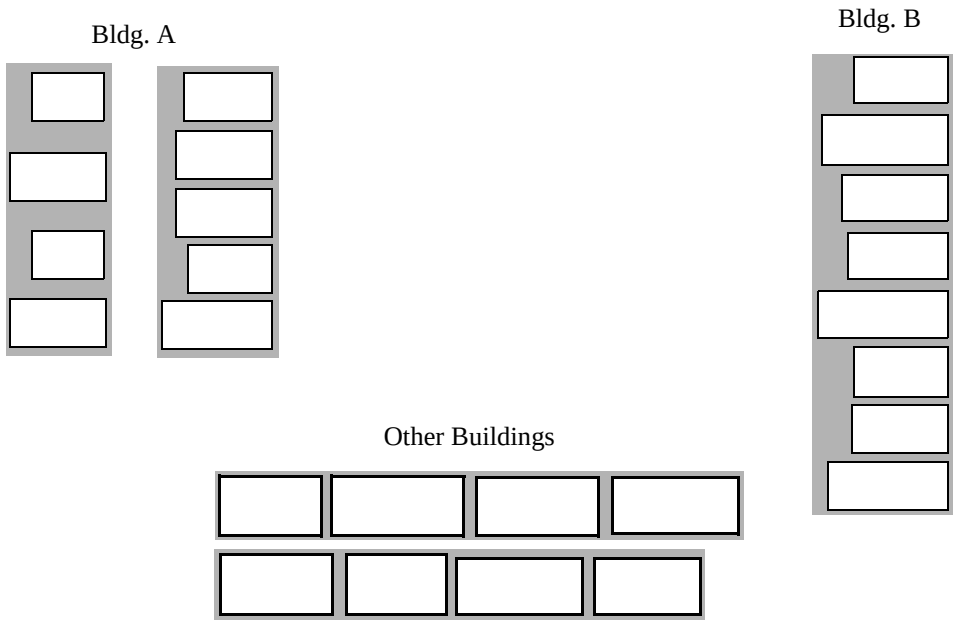

Figure 7-2. Desired groupings of diagram elements

allowed us to compare, for each subject, whether the broom or standard commands were better for the task. Ten subjects each repeated this with three separate diagrams. One of these diagrams is shown in Figure 7-2.

On the second and third diagrams, we tested the short-term memory of our subjects to see if the memory load was greater for one tool than for the other. Before each diagramming task, subjects memorized a set of six random, two-digit numbers, and at the beginning and end of each task they were asked to recall the numbers. The expectation was that more numbers would be forgotten when using a tool that requires more short-term memory to use. This technique for measuring short-term memory load was inspired by a recently published experiment (Byrne and Bovair, 1997).

Results. In all thirty trials, the mouse was moved a greater distance when using the standard tools than when using the broom. On average, the mouse was moved 86% farther

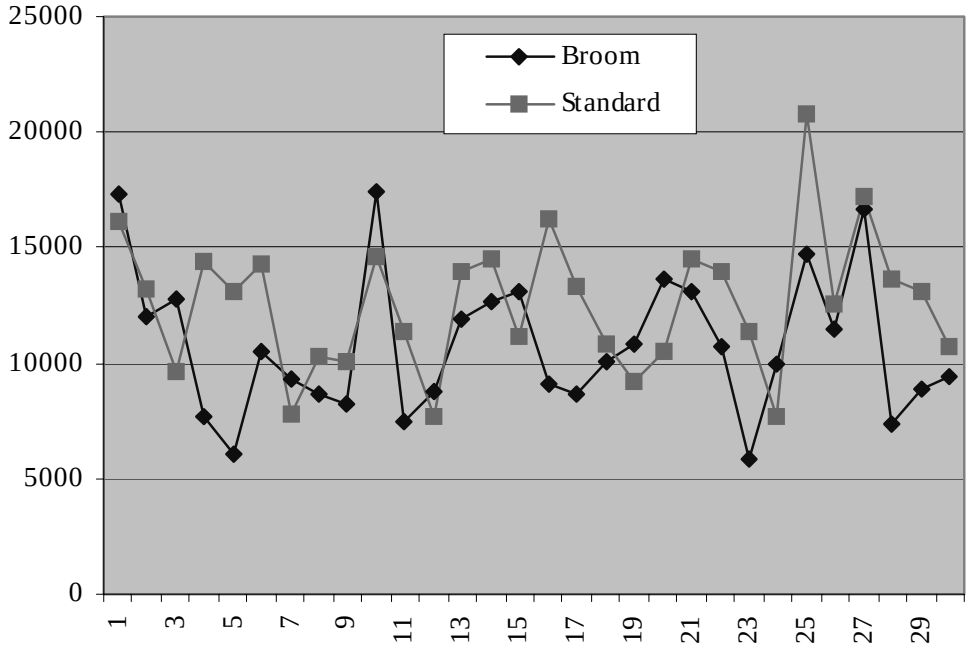

Figure 7-3. Mouse dragging with the broom or standard alignment tools

when using the standard tools. This was largely due to movement to a toolbar of alignment buttons at the top of the drawing area. In contrast, control-drag was used to invoke the broom. This difference would be reduced if keystrokes were assigned to each alignment command; however, that would require eight new keystroke bindings and may force users to move their hands between the mouse and keyboard more.

Figure 7-3 charts the distance that subjects dragged the mouse. Since the broom involves dragging the mouse and dragging can be relatively difficult, we were concerned that the broom might be more physical tiring. However, over all trials, subjects dragged an average of 12,592 pixels while using standard tools and only 10,809 while using the broom, which is 16% shorter. Using a paired t-test, we found the difference to be significant with $P < 0.003$. A large part of the dragging needed for standard tools was

done while dragging out selection rectangles. The shorter dragging distance for the broom resulted largely from the fact that objects do not need to be explicitly selected before they are aligned with the broom.

Achieving layouts that show grouping and correspondence requires planning: performing alignments in the wrong order can force users to undo previous work. Since using the broom involves fewer planned steps, we expected a lower short-term memory load when using the broom. In fact, the majority of subjects indicated that they found the broom more “natural.” However, we found no significant difference in the short-term memory effects of the tools compared. We believe that our test for short-term memory load was not sensitive enough to detect the differences between the tools. In fact, subjects recalled all numbers perfectly in twenty-six out of forty tasks. This led to a refined version of the short-term memory load test in the next study.

7.3 Construction User Study

Goals. The goal of this user study was to evaluate the support provided by Argo/UML’s selection-action buttons. In particular, this study focused on measuring the match between designers’ diagram construction tasks and the user interface affordances provided by selection-action buttons.

Setting. This study was approved by the UCI human subjects research committee as study HS99*1210 and was carried out in August 1999. The study consisted of five subjects performing prespecified diagram construction tasks under two conditions. Under one condition, subjects used standard diagram construction toolbar buttons. Under the

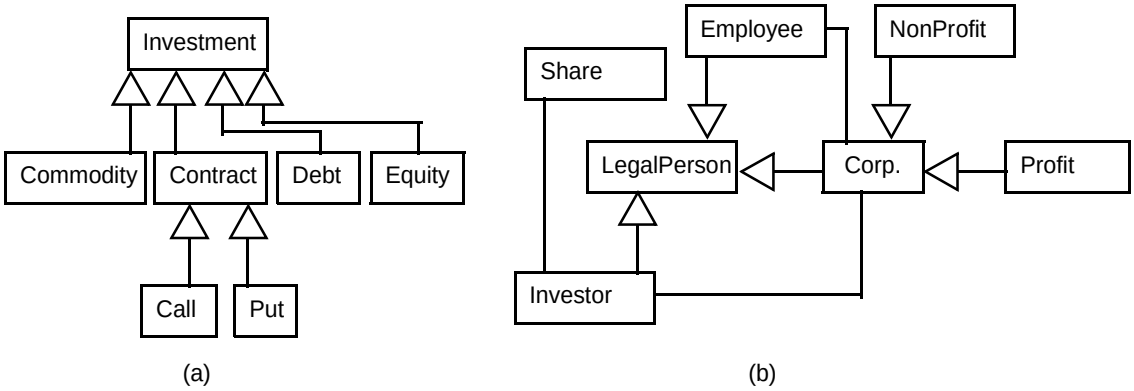

Figure 7-4. (a) Conventional diagramming task used in selection-action button study, (b) Unconventional diagramming task

other condition, subjects were encouraged to use the selection-action buttons for most of the construction. As in the earlier broom study, each subject was asked to do each task twice: once under each condition, in random order. Short-term memory load was also measured using a refinement of the technique used in the broom study.

Results. The subjects in this study were all able to accomplish the diagramming tasks with either the selection-action buttons or the standard toolbar. Only one subject had a difficult time with the selection-action buttons. The primary difficulty was in making the buttons appear rather than actually using them. Surprisingly, many of the subjects formed mistaken assumptions about the action needed to get the selection-action buttons to appear. Also, all but one of the subjects used each of the three aspects of the selection-action buttons. Since the task was a transcription task rather than a design task, subjects tended to work systematically from upper-right to lower-left rather than expanding on logical clusters.

Subjects made more mistakes on the unconventional diagramming task than on the conventional one. For example, several subjects mistakenly used association links rather

than horizontal generalizations. No subject, however, made the same mistake more than once, and several said that they understood that the directions of the lines were "misleading" on this task. Bent edges caused users to click on the first vertex of the edge, resulting in placement of a new node. Subjects also made several mistakes when using the standard toolbar buttons, including using the wrong type of edge and forgetting the current mode. Some users double-clicked in the toolbar to lock in a node creation mode and then accidentally created extra nodes while attempting to reposition existing nodes.

The study results suggest three possible refinements to the selection-action button feature. First, the buttons should appear and stay visible whenever a node is selected, even if the node has been slightly moved during selection or if the mouse has been moved away from the node. Second, when dragging to specify the location of a new node, the mouse coordinates should be used as one corner or the center of an edge rather than the center of the node.Third, the dragging behavior of selection-action buttons might be modified to allow the creation polygonal edges by clicking in empty space to add a vertex and using double-click in an empty space to create a new destination node.

7.4 Classroom Usage

Goals. The goal of this empirical evaluation is to gather anecdotal experience from classroom usage at UCI and other universities. One of my reasons for developing Argo/UML was to produce a freely available tool that could be used in university classrooms and that would help teach good design. Actually aiding the *teaching* of software design skills requires much more than simply using an educational license for a standard commercial UML tool. Argo/UML's strong basic usability and cognitive support features

can help address the needs of users who are new to UML and object-oriented design. In addition to actually using Argo/UML, several students have made contributions to Argo/UML's development as part of project courses, independent study courses, or research projects.

Setting. Argo/UML, Argo/C2, and GEF have been used in several courses at UCI and at other universities. Table 7-1 summarizes classroom usage of these tools. The uses outside of UC Irvine were found by searching for "Argo/UML" and "GEF" on popular

web search engines and by reviewing email messages sent from university users of GEF and Argo/UML.

Table 7-1: Known classroom usage of the Argo family

School	Date	Course	Usage
UC Irvine	Summer 1999		Student research project
UC Irvine	Spring 1999	ICS 227	Integrated Argo/C2, Argo/UML, and other tools
UC Irvine	Spring 1999	ICS 125	Enhanced Argo/UML
UC Irvine	Spring 1999	ICS 121	Class used Argo/UML
UC Irvine	Fall 1998	Independent study	Enhanced Argo/UML
UC Irvine	Fall 1998	ICS 125	Class used Argo/UML
UC Irvine	Winter 1998	ICS 125	Enhanced GEF
UC Irvine	Winter 1998	ICS 125	Enhanced Argo/UML
UC Irvine	Fall 1996	ICS 125	Developed initial version of GEF
U. Twente, The Netherlands	1999		Student research project
U. Waterloo, Canada	1999		Student research project
U. Mulhouse, France	1999		Survey of UML tools
U. Frieberg, Germany	1999		Student research project
U. Vrije, Brussels	1999		Master's thesis
U. Bologna, Italy	1999		Master's thesis
Oregon State	Spring 1999	CS 562	Grad seminar discussion topic
Syracuse U.	Spring 1999		GEF used in research project, mentioned in paper at Supercomputing '98
UC Berkeley	1999		Research project inspired partly by GEF
U. Macow, Macow	1998		Student project used GEF
UCLA	Winter 1997		GEF used in research project
Duke U.	Winter 1997		Student project used GEF
Syracuse U.	Fall 1997	CSP714	GEF used in project course
Purdue U.	Summer 1997		GEF used in student research project
North-Eastern U.	Summer 1997		GEF used in student research project
CMU	Summer 1997		GEF evaluated for research project

Results. Classroom usage of Argo/UML at UC Irvine has been a practical success in that students have been able to complete their assignments. In the Fall 1998 offering of ICS 125, individual students reported lost data, slow performance on machines in their

team offices, and a few basic usability problems. None of the students reported dissatisfaction with the cognitive support features, which probably indicates that they were not frequently used. However, one of the goals of cognitive support is not to interfere with basic tool usage, and this goal seems to have been achieved.

In the Spring 1999 ICS 121 course that used Argo/UML, students were able to begin using the tool rapidly and complete their assignments. Very few problems of any kind were reported, and those that were reported mainly requested new functionality (e.g., a special kind of cut and paste) or environmental difficulties (e.g., printing in the CS 3rd floor lab).

Student projects that enhanced GEF and Argo/UML have generally been successful. Three ICS 125 projects have enhanced Argo/UML with new diagram types, and each of these projects has produced some code that is worthy of incorporation into the distributed version. Students involved with the projects have generally reported a feeling of satisfaction that they were contributing to a project that other students would continue to evolve and use. Undergraduate independent study course have produced several contributions, including a key part of Argo/UML's support for the XMI file format. Research projects by UCI graduate students have built on code in the Argo family to produce ArchStudio and Argus-I.

7.5 Internet Usage

Goals. One of the goals of my research is to have an impact on what CASE tool users expect from their tools and what CASE tool vendors provide. Research on design

critiquing systems and other forms of cognitive support has been carried out for over fifteen years, yet most users of desktop applications have never encountered a system with critics. I have tried to transfer my ideas from the research environment to industry by demonstrating them in the context of a useful tool. As discussed below, measurable progress on this goal has been made, but it is far from complete.

Setting. This section presents anecdotal and statistical data on usage of Argo/UML by internet users. Here, "internet users" refers to people who downloaded Argo/UML and who are not educational users. These users were self-selected and their feedback was voluntary. Most of these users found the Argo/UML web site by searching the internet for the terms "UML" or "CASE", while others learned of the site from other users.

Whenever anyone downloads Argo/UML, they enter registration information that includes their email address. From July 1998 until January 1999, I followed up on each registration by sending an email message that stated "Thank you for your interest in Argo/UML," and asked "What is your interest in CASE tools?" After initial contact, I continued to receive feedback from many users. Since Argo/UML is not yet fully developed, users have often encountered difficulty that prompted them to ask questions, offer comments, or report bugs. This data is also voluntary and comes from self-selected subjects.

Results. The first and most surprising result is that Argo/UML's registered users include thousands of people from all around the world. This indicates that Argo/UML is

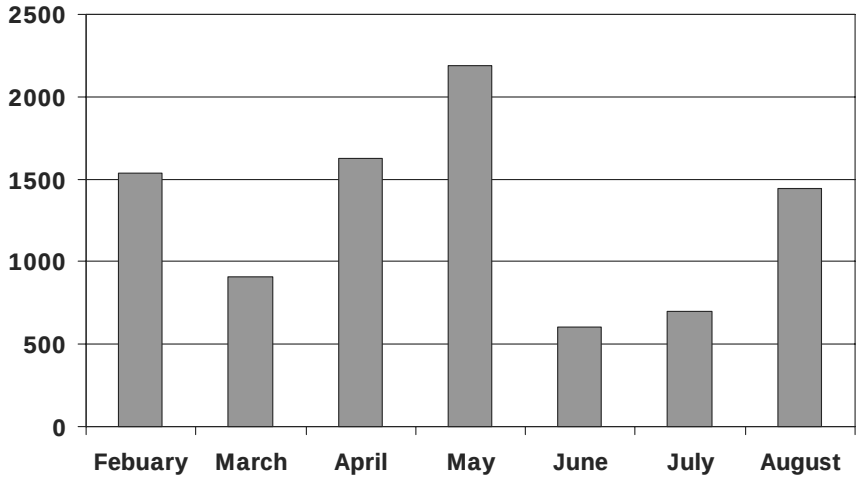

Figure 7-5. Number of new Argo/UML registered users by month in 1999

at least reaching many of the potential CASE tool users whose expectations I seek to raise. Figure 7-5 shows the number of new registered users each month.

One reason that so many users found the Argo/UML web site is that it is listed in many search engine databases and CASE tool index web pages. Searching for “Argo/UML” on leading internet search engines yields over one hundred hits on pages outside of UCI. These sites are hosted in well over a dozen countries, and they typically offer links to the Argo/UML home page and brief descriptions of the tool in English, German, French, or Japanese.

Between July 1998 and August 1999, I have received a total of one-hundred twenty-five bug reports on Argo/UML. The number of bug reports helps to define a lower bound on the number of people who have actually used Argo/UML. Typically, only one, two, or three bug reports are submitted by the same person, and a total of seventy-four distinct

people have submitted bug reports. Of course, many more people have used Argo/UML without submitting any bug reports.

Table 7-2: Some quotes from Argo/UML users

My company bought a CASE tool called ... COOL. Of course, users had less polite names for it. I'm not sold on CASE since I feel you spend more time working the tool than doing design, but I'm still open.
I was showing Argo/UML to my boss, and we were both impressed with the design guidance features. The idea of the checklists and critiques, especially with the future possibility of tailoring them for a user's specific strengths and weaknesses, seems to add to potential reliability of the produced models.
The selection-action buttons are really fantastic time-savers. I can't believe that no on thought of them before.
I have used Object Domain, PepperSeed,... I have never found a really good UML tool, so I am always on the lookout for... ease-of-use and speed.
I've used ObjectTeam, COOL, Rose, P-Plus. I think they are all detractive to UML. I'll try Argo/UML, as it seems to have a nice look and feel!
Argo/UML looks like a very promising product. I'm a software engineer currently using Rose. I am exploring alternative tools because it is too expensive to have a copy at home, and using it can be frustrating.
I especially love the critiques that pop up when the mouse hovers over a diagram element. Table views are an extremely nice touch.

Table 7-2 presents some excerpts from email messages sent from Argo/UML users. Feedback from users has been uniformly positive about Argo/UML's cognitive support features. I have received very few negative email messages. One negative message complained about the need to provide one's name and email address during registration. Some other negative messages basically said that Argo/UML was not ready for use in their organization. I believe that the later type of comment occurred because Argo/UML's overall impression of commercial quality can lead users to forget that it is a research project.

The following observations are drawn from 783 email messages I received from 602 distinct Argo/UML users:

- As with classroom usage, the majority of user complaints and bug reports identified missing functionality or outright implementation defects. No one has ever reported that design critics or any other cognitive support feature has interfered with their work.

- Many comments came from sophisticated CASE tool users who had experience using several commercial tools, but more came from first time users who were starting to learn UML and object-oriented modeling. Comments from experienced CASE tool users often focused on the perceived difficulty of using tools like Rational Rose. Usability and subjective satisfaction seem to be key factors in whether CASE tools become shelfware. Developers new to modeling often stated that they were the first person in their development organization to use modeling and that they saw Argo/UML as potentially helpful in learning UML.

- The zero cost of Argo/UML generated more enthusiasm than did the cognitive support features. Emphasis on cost is somewhat unexpected since CASE tool users typically do not pay for the tools themselves. Also, most software designers are highly paid, so marginal increases in productivity could result in savings much greater than the initial cost of the tool. Furthermore, much of the cost of CASE tool adoption is in training rather than tool price (Huff, 1992). All of these reasons should make CASE tool purchasers insensitive to prices. In fact, that price assumption is reflected in the state of the CASE tool market where tools typically cost $2000 to $6000 per seat. However, given the other user comments on dissatisfaction with CASE tool usability, it seems that price sensitivity may stem from the broader perception that CASE tools are not worth using.

One very interesting segment of the Argo/UML user population is made up of people who are employees of CASE tool vendors. I have had received email from employees of Rational (makers of Rose), Togethersoft (makers of Together/J), MicroGold (makers of WithClass), Object Insight (makers of JVision), and ObjectDomain (makers of ObjectDomain). In four out of five cases, these email contacts have been with lead designers or company presidents. For the most part, the email messages have briefly expressed interest in Argo/UML without stating any specific views on its cognitive support features. However, there have been two cases where Argo/UML features have influenced the features of commercial tools.

CHAPTER 8: A Scalable, Reusable Infrastructure

Software development tools are complex software systems that are difficult to build. CASE tools must include detailed design representations, sophisticated user interface elements, and bindings to specific programming languages. These tools are difficult to build in a research setting because of the effort and breadth of understanding required. Yet, research on CASE tools is needed to produce better and more usable tools. One of the goals of my work is to provide an infrastructure for further development of CASE tools in research settings.

My approach to providing a CASE tool infrastructure consists of three main elements: (1) I developed frameworks for several aspects of the CASE tool application domain; (2) I demonstrated the use of these frameworks in a useful CASE tool; and, (3) I organized an open source development project that has brought together and facilitated the work of researchers interested in CASE tool development. This chapter explains the frameworks that make up Argo/UML's infrastructure.

"A *framework* is a collection of abstract and concrete classes and the interface between them, and is the design for a subsystem" (Wirfs-Brock and Johnson, 1990). Abstract classes are classes that provide only a partial implementation of the interface that they declare. Each abstract class must be subclassed with concrete classes that completely implement the declared interface. Frameworks embody knowledge about the application domain and outline an appropriate implementation while still avoiding commitment to particular implementation details. Pree (1995) emphasizes that every reusable software

artifact consists of parts that remain constant across uses and parts that are adapted to a particular application. In each of the frameworks discussed below, I have kept a clear distinction between the abstract classes in the framework that provide reusable infrastructure and the concrete classes in Argo/UML that specialize that framework.

I chose the framework approach over several alternative approaches described in the literature (e.g., Krueger, 1992) because class frameworks present a low barrier to reuse: the framework approach is well known to many object-oriented software designers and requires no special tool support. The result has been widely successful in the case of my graph editing framework. I have also successfully reused my Argo critiquing framework in several design tools. The frameworks for navigational perspectives, checklists, and code generation have yet to be reused outside of Argo/UML.

8.1 Graph Editing Framework

8.1.1 Introduction

All of the diagram display and editing features in Argo/UML and Argo/C2 are implemented as part of a reusable Java framework called GEF (Graph Editing Framework). Work on GEF began in Spring 1996 and the first version was released via a web site later that year. Since then, GEF has evolved substantially and has been used in dozens of research projects. GEF currently consists of about 24,000 lines of Java code in 160 classes. Argo/UML's implementation of five diagram types (class, state, use case, activity, and collaboration) extends GEF with an additional 10,000 lines in 56 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 (Beck and Johnson, 1994) and Unidraw (Vlissides and Linton, 1990). GEF takes this previous work into account but emphasizes extensibility, simplicity, and a high-quality user experience. HotDraw and Unidraw both achieve great extensibility by using flexible, abstract concepts. I limited the number and flexibility of GEF's concepts to make the framework more understandable. Over time, GEF has been applied to many diagram types and its look and feel provide a better user experience, but these extensions have not required a generalization of GEF's basic concepts.

8.1.2 Design Overview of GEF

Figure 8-1 gives an overview of GEF's design. There are six major concepts in GEF. (1) The Editor class acts as a mediator that holds the other pieces together and routes messages among them. (2) Figs (short for figures) are the primitive shapes; for example, FigCircle draws a circle and FigText draws text. (3) Layers contain Figs in back-to-front order. (4) 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. (5) 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. (6) 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. I 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 (Gamma et al., 1995) or attempt to offer general purpose constraint solving (e.g., Sannella, 1994).

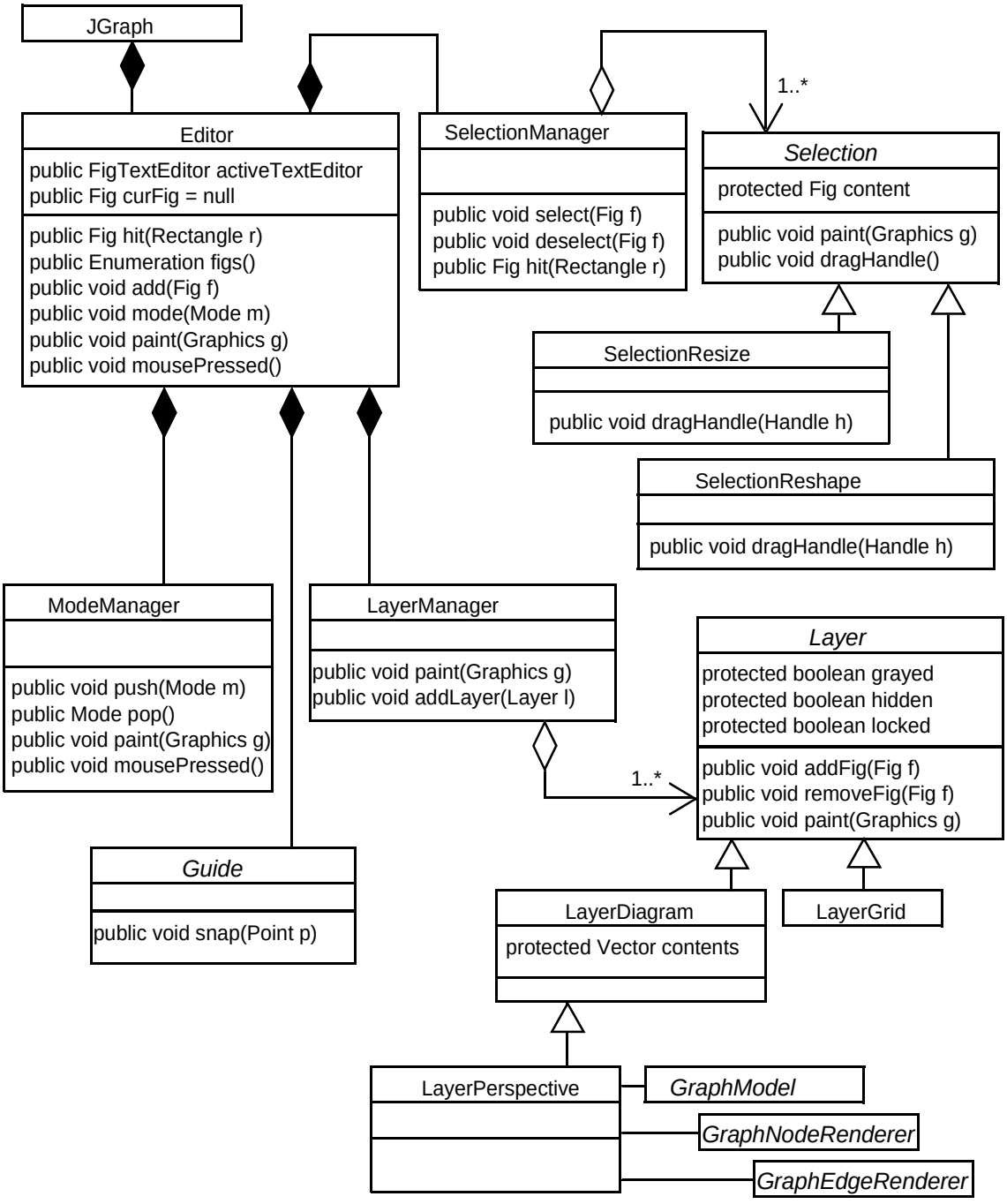

Figure 8-1. UML class diagram of GEF

Initially, I implemented a generic connected graph representation as the underlying model for GEF diagrams. After using GEF in several applications, I found that most applications have existing data structures that can be interpreted as graphs. For example, Argo/UML has the UML meta-model and Prefer had its own data structure to represent state-based requirements. In the revised GEF, GraphModels manage the mapping from Figs in a diagram to application objects in an underlying data structure. GraphModels themselves do not hold much data; rather, they interpret existing data structures as graphs. For example, StateDiagramGraphModel interprets UML States as nodes and UML Transitions as edges. GEF's 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 to interpret underlying data structures as trees or tables.

Rather than represent diagrams as nodes and edges, GEF represents diagrams as ports, nodes, and edges. Ports are connection points on nodes, and edges go from a source port to a destination port. The inclusion of ports in the graph model was inspired by my previous experience with OBPE (Robbins et al., 1996). Some diagram types assign semantics to the point where an edge meets a node. Ports allow GEF to represent the semantics of these diagram types. For example, Argo/C2 represents a C2 component as a node and the top and bottom interfaces of the component as two ports. Most UML diagrams do not assign meaning to the point where an edge meets a node. In these cases, Argo/UML uses a single, invisible port that is the same size and shape as the overall node.

8.1.3 Implementation of Multiple Diagrammatic Views

Like many user interface systems, GEF loosely follows the Model-View-Controller (MVC) design pattern to support multiple views (Krasner and Pope, 1988; Gamma et al., 1995). As with many MVC implementations, GEF sometimes combines the view and controller roles into the same object. GEF's GraphModels play the role of the model: they provide access to the semantic state of the diagram and send notification messages when that state changes. Layers and Figs act as models for the visual properties of the diagram, including coordinates, colors, and back-to-front ordering. GEF's Editors, Layers, and Figs provide most of the functionality of the view role by displaying the diagram. Other view functionality is provided by GEF's Modes and Selections, which also contribute graphics for interaction feedback. GEF's Modes and Selections primarily play the role of controller; however, GEF provides the option for Figs and model elements to perform some event handling.

GEF uses composition structures for models, views, and controllers. CompoundGraphModel is an abstract class that combines simple graph models into more complex ones. Layers and FigGroups (including FigNodes) are compositions of views (i.e., Figs). GEF's ModeManager is composed of several controllers (i.e., Modes). The ModeManager is unusual in that it maintains a stack of active Modes rather than a single current Mode. Each Mode in the stack is asked to handle an incoming event until one of them successfully handles it. The last-in-first-out nature of the stack supports temporary Modes, such as ModeBroom, that are pushed onto the stack to process events in a short interaction and then popped off.

The MVC design pattern is an extension of the Observer-Observable design pattern (Gamma et al., 1995). In MVC, the model plays the role of an observable object that sends notifications of changes, and the view plays the role of an observer that reacts to these change notifications. GEF uses the Observer-Observable pattern at three levels:

- GEF's Figs and Layers act as observables for Editors to observe. This allows multiple diagram windows to show the same diagram. Multiple windows on the same diagram are not often used, but carefully implementing this feature avoids potentially confusing inconsistencies when multiple windows are used. For example, a designer using Argo/UML may create a new view of a diagram by double clicking on the "As Diagram" tab. This displays the diagram in the main pane of the Argo/UML window and also in a new, larger diagram window.

- GEF's GraphModels act as observables for GEF's Figs and Layers to observe. This allows for multiple diagrammatic views of the same connected graph. Each of these views may show a different projection of the connected graph and may use a different notation. For example, one Argo/UML class diagram may show some of the classes in a given package and another may show a partially overlapping set of classes in the same package. Argo/UML does not take advantage of GEF's ability to use different kinds of Figs to present the same GraphModel elements, but that feature could be used to support alternative notations such as OMT (Rumbaugh et al., 1991), or the Booch Notation (Booch, 1991).

- An application's underlying data structures (e.g., Argo/UML's implementation of the UML meta-model) act as observables for GEF's GraphModels to observe. This allows the same design representation to be viewed and edited in very different ways. For example, the dual of a graph could be edited via an alternative GraphModel that interprets data structure elements as edges rather than nodes and nodes rather than edges.

8.1.4 Implementation of the Broom Alignment Tool

The broom alignment tool is a fairly straightforward extension to the basic GEF classes. The broom is implemented as a subclass of class Mode that interprets mouse movements and keystrokes as commands that change the position of the broom and move

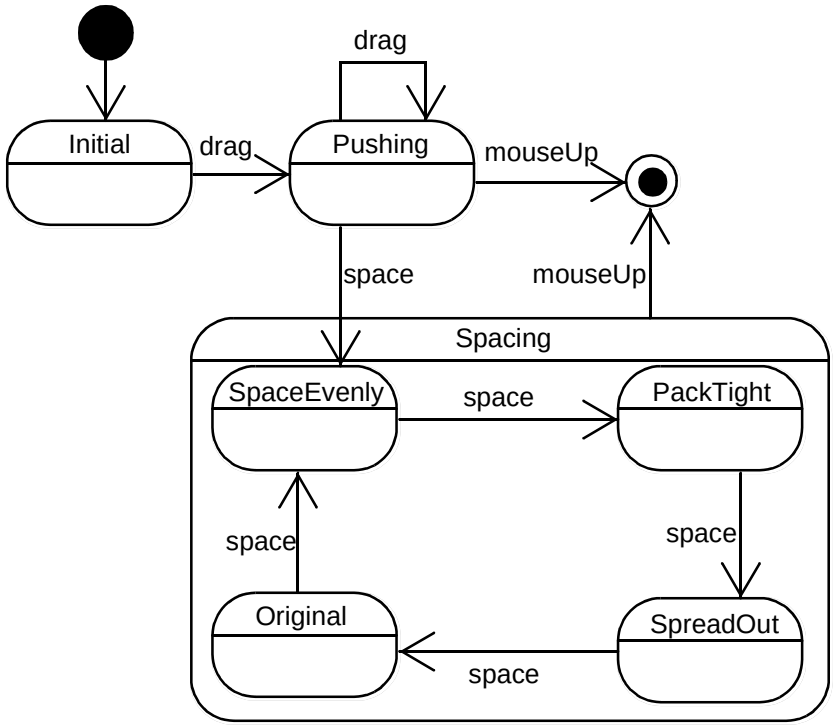

Figure 8-2. Broom states

graph nodes that are on the face of the broom. A state machine for the broom mode is shown in Figure 8-2, and each state is explained in Figure 8-1.

Table 8-1: Description of broom states

State Name	Description
Initial	Plus-sign drawn in blue. On drag, direction determined.
Pushing	Broom drawn in blue. On drag, update touching list, move each touching object to broom position or original position, expand broom size if dragged laterally.
Space-Evenly	Broom drawn without tail. Space touching objects evenly within their original bounding box.
Pack	Broom drawn without tail. Store original location of touching objects. Space touching objects with fixed gaps starting from left or top of broom.
Spread-Out	Broom drawn without tail. Space touching objects evenly along the length of broom.
Original	Broom drawn without tail. Return touching objects to their stored original location.

8.1.5 Implementation of Selection-Action Buttons

Selection-action buttons are implemented as a fairly straightforward extension to GEF's Selection classes. Class SelectionWButtons implements the display and click-or-drag behavior of all selection-action buttons, while the subclasses of SelectionWButtons define the buttons that are appropriate to each particular type of object.

The selection-action buttons are treated the same way that resizing handles are treated: when ModeModify detects that the user has clicked or dragged on a handle it sends a message to the SelectionWButtons object asking it to process the event. When a selection-action button is clicked, a new node is created along with an edge between the current node and the new node. When a selection-action button is dragged, a new ModeCreateEdgeAndNode is created and pushed onto the ModeManager's stack. ModeCreateEdgeAndNode draws a rubberband line while the user drags and then creates a new edge and possibly a new node when the mouse button is released.

8.2 Argo Kernel

8.2.1 Introduction

The Argo kernel is a class framework that provides infrastructure for knowledge support features in applications such as Argo/C2, Prefer, and Argo/UML. An earlier Smalltalk-80 version of this critiquing framework was used in the Stargo OMT tool and the initial version of Argo/C2. The framework focuses on representation and algorithms that support design critics, criticism control mechanisms, checklists, the dynamic "to do" list, clarifiers, non-modal wizards, design history, and a user model.

The Argo kernel currently consists of just under 5000 lines of Java code in 34 classes. Argo/UML specializes the Argo kernel with classes for 74 critics, 6 wizards, and 3 clarifiers totaling 8300 lines of Java source code. Argo/UML also defines 11 checklists totaling just over 1000 lines.

8.2.2 Design Overview of the Argo Kernel

Figure 8-3 gives an overview of the Argo kernel's design. The key classes are Designer, Agency, Critic, ControlMech, ToDoItem, ToDoList, and History. A single instance of class Designer represents the user of the design tool; it includes identifying information, preferences, a ToDoList, and a user model. Class Agency is a utility class that provides methods for scheduling and applying critics. Design critics are instances of class Critic, while criticism control mechanisms are instances of class ControlMech. Argo's critic scheduling algorithm involves the Agency, critics, and control mechanisms; it is described in the next section. Critics produce ToDoItems when they detect design improvement opportunities. ToDoItems are stored in the designer's ToDoList. A single instance of class History stores a time-ordered list of HistoryItem objects; a given HistoryItem object can store information about a criticism that was raised, a criticism resolution, or a design manipulation.

Class Wizard and interface Clarifier enhance the basic Argo kernel by providing reusable infrastructure for non-modal wizards and clarifiers, respectively. Clarifier is an interface, i.e., a Java language construct similar to a class, but consisting only of method declarations without bodies. Clarifier builds on a standard Swing interface for displaying icons and adds methods to detect when the mouse is over the clarifier. Clarifiers are

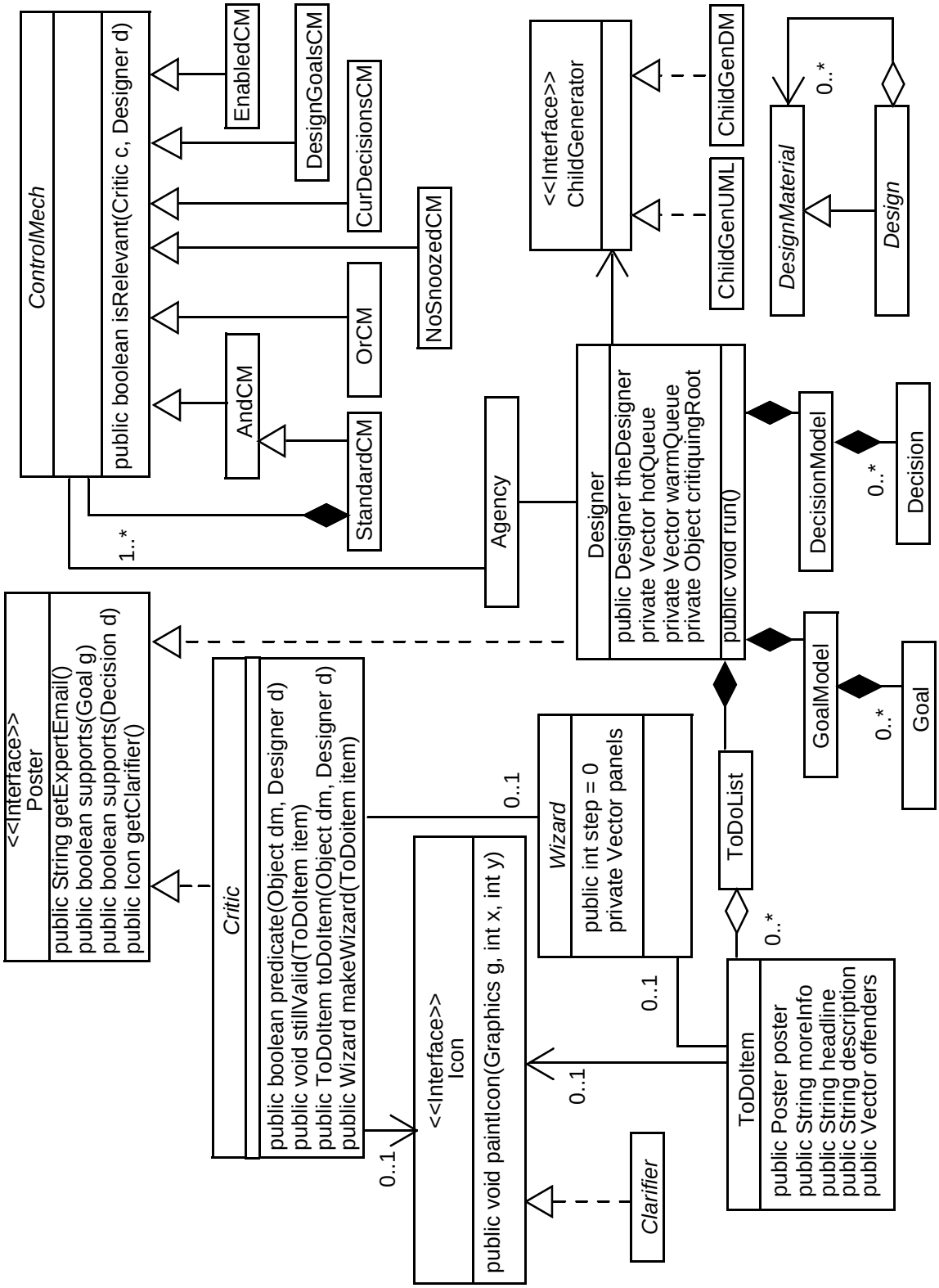

Figure 8-3. Classes implementing the Argo kernel

implemented using the Singleton pattern to avoid the overhead of being allocated and freed each time new clarifiers are needed (Gamma et al., 1995). Since they are shared singletons, clarifiers hold no long-term state information about the design element or ToDoItem that they present, and instead, only keep that information for the duration of each display operation. Making clarifiers singletons complicates their design somewhat, but it helps maintain good interactive performance by reducing the time needed to display the currently selected design element.

The Argo critiquing framework provides a default design representation that can be used if the application does not already have a design representation data structure. Instances of class DesignMaterial represent the design elements, while instances of class Design represent the overall design or meaningful subsections. DesignMaterial instances store a dictionary of named properties and a list of observer objects needed for the DesignMaterial to play the role of observable in the Observer-Observable design pattern (Gamma et al., 1995). Applications can use class DesignMaterial directly, or make specialized subclasses if needed. In the initial version of the Argo critiquing framework, these classes played a major role and their use was required. But now, the default design representation is not used in the Prefer and Argo/UML tools because these tools have their own design representation data structures. When existing data structures are used, the application must supply a class that implements the ChildGenerator interface to allow the critiquing routines to walk the design representation. Also, existing data structures may send notification of state changes to help focus critiquing; however, critiquing will still work without change notifications, albeit more slowly.

8.2.3 Implementation of Design Critics and Criticism Control Mechanisms

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 declares relevant decision categories. The main method is predicate() which 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 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 a previously produced "to do" item should now be removed from the "to do" list because, e.g., the identified problem has been resolved.

Criticism control mechanisms are also implemented as Java classes that implement a predicate function. However, in this case, the predicate accepts a critic rather than a design element 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, otherwise, it is disabled.

The scheduling and application of critics operate within a critiquing thread of control so as not to interrupt or delay 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," and (3) applying critics to a few

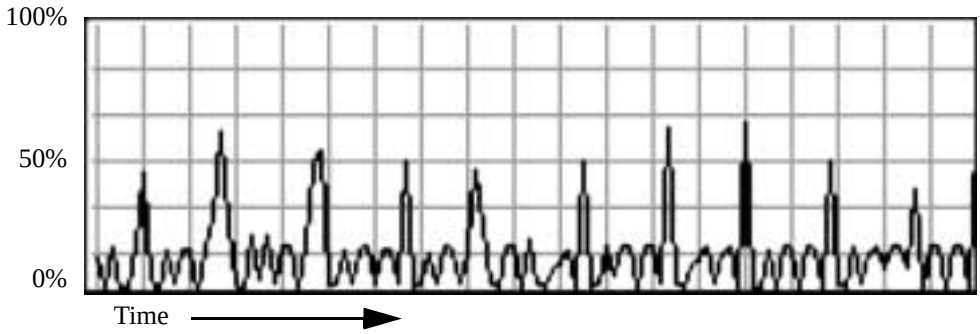

Figure 8-4. CPU load imposed by critics on a 233MHz computer with Windows NT

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 all but the smallest design projects, this traversal takes much longer than the desired interactive response time of about one half of a second. The hot queue contains only design elements that are likely to generate new feedback and is typically short enough to be completely cleared within half a second. Design elements are promoted to the hot queue in response to design manipulations that have the possibility of introducing problems. Application of critics to elements of the hot queue is further focused by applying only those critics that are registered as having interest in the type of design manipulation that promoted the design element.

One key trade-off in a critic scheduling algorithm is the amount of knowledge the scheduler has about each critic. With no knowledge as to 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 keep the scheduler simple and reduces the development effort needed to add a critic. Providing more knowledge about critics allows the critiquing system to work more efficiently and reduces application times. The Argo critiquing framework requires that all critics register interest in specific types of design elements and allows critics to register interest in specific types of design manipulations. If the critic author chooses not to specify which design manipulations should trigger the critic, or does so incorrectly, the critic will still be applied eventually.

The approach to implementing critics described above is somewhat similar to the way expert systems are implemented (Lee, 1990; Subramanian and Adam, 1993). I have chosen a set of trade-offs based on experience with building and using design support systems. My approach also allows critic authors to use a standard programming language rather than a limited rule language. Critics are allowed to have their own state, arbitrary side effects, and may even invoke native executables or communicate with external servers. This allows critic authors to repackage existing analysis tools as critics. In contrast, most expert system rule languages or constraint languages do not offer these possibilities. One possible extension to this framework would be to extend the general critiquing framework with specific support for a rule or constraint language, such as OCL (Object Constraint Language) (Warmer and Kelppe, 1999).

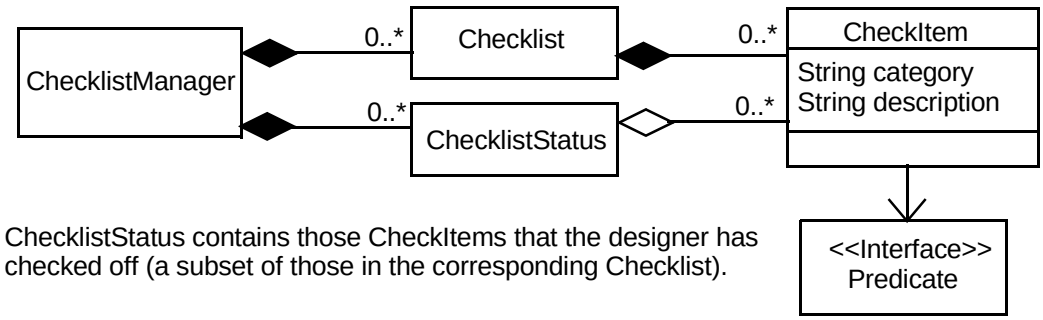

Figure 8-5. UML class diagram of Argo checklists

8.2.4 Implementation of Checklists

Argo's implementation of checklists is somewhat similar to its implementation of design critics, however, checklists are much simpler. The key classes implementing checklists are shown in Figure 8-5. The CheckManager is a utility class that keeps track of all known checklists and the status of each. The checklist viewer uses the CheckManager to find the checklist that is appropriate for the currently selected design element. An instance of class Checklist contains a collection of CheckItems. Each CheckItem instance has a category string, a description string, and a guard predicate. The predicate object implements a method predicate() that determines whether that CheckItem will be displayed to the user. ChecklistStatus instances keep track of which items in a checklist have been marked as already considered by the designer.

Unlike critics, checklist items provide more general advice and they are almost always displayed to the user; i.e., their guard conditions are almost always true. The evaluation of checklist item guard conditions is not scheduled, instead it is done whenever the designer changes the currently selected design element. A possible improvement would be to only evaluate those guards for checklists that are visible on the screen.

8.2.5 Implementation of Wizards

Class Wizard is an abstract base class for non-modal wizards. It provides some code to implement features common to all wizards and declares some methods without providing method bodies; actual wizards must implement these methods with code specific to each wizard.

All non-modal wizards consist of a set of user interface panels that are constructed as needed when the designer presses the “Next>” button to move on to the next step of the wizard. By convention step zero is the problem description of the ToDoItem, step one is the first panel displayed after the user presses “Next>”, and so on. The problem description panel is not stored in this wizard, only the panels that are specific to the wizard are stored. This allows for the designer to rapidly browse many “to do” items without incurring the overhead of creating wizard panels that may never be used.

Specific wizards are implemented as subclasses of class Wizard. These subclasses construct user interface panels for each step of the wizard as needed, implement predicates that enable or disable the “Next>” and “Finish” buttons, and implement the actions to be taken when the designer moves from step to step. On each forward step (the “Next>” button), class Wizard calls doAction() if the current step has never been completed before, and redoAction() if the designer has backed up and is now moving forward through a previously completed step. On each backward step (the “<Back” button), class Wizard calls undoAction() to reverse the effects of a previous doAction() or redoAction(). Specific wizards must implement doAction(), redoAction(), and

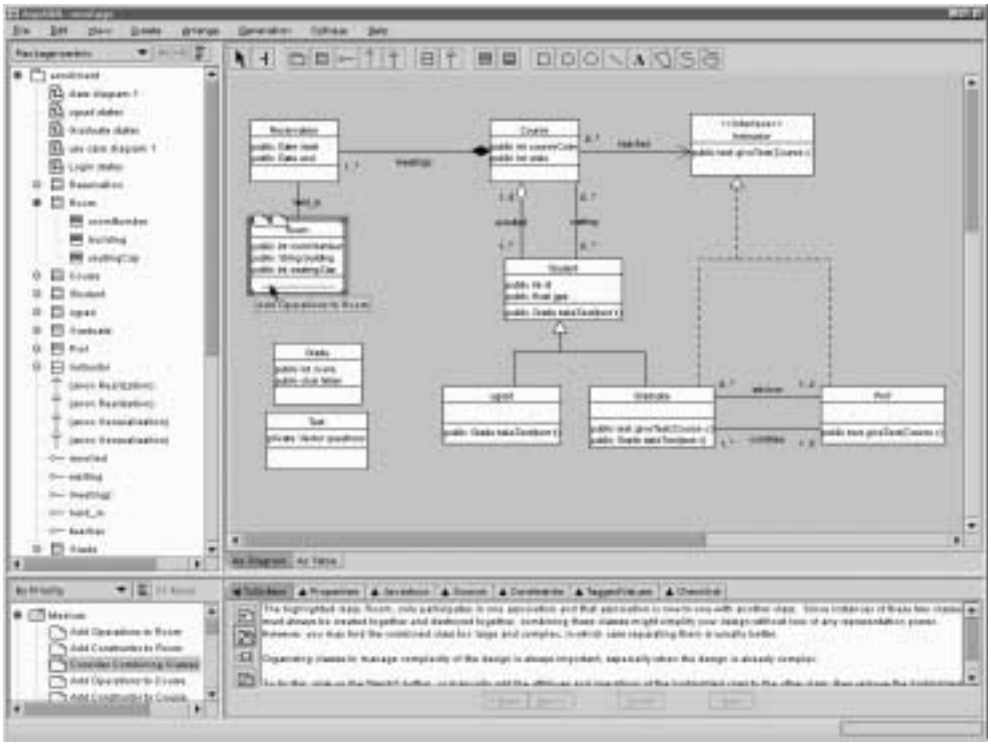

Figure 8-6. Argo/UML main window

undoAction() as appropriate, but they do not need to implement the logic that maps buttons to actions or determines when a step has previously been completed.

8.3 Views and Navigation

8.3.1 Introduction

Figure 8-6 shows the Argo/UML main window. This section discusses the implementation of several of Argo/UML's non-diagrammatic design views.

A recurring theme in the implementation of Argo/UML's design views is the use of the Model-View-Controller design pattern (Krasner and Pope, 1988; Gamma et al, 1995) where the view and controller roles are played by a user interface widget, and the role of the model is played by a mediator class. These mediator classes define task-specific views

of the underlying design representation and contain very little state information themselves. The use of mediator classes is one of the standard ways to use the Swing user interface library (Eckstein, Loy, and Wood, 1998). Mediator classes observe the design representation and react to change notifications by sending their own change notifications that cause the view to be redrawn.

The Argo/UML source code for views and navigation consists of about 30,000 lines of code in 193 classes. The majority of that code is reusable infrastructure in the form of base classes and utility classes; the remainder of the code specializes the reusable infrastructure to the UML meta-model and the object-oriented design task.

8.3.2 Design Overview of Argo/UML Views and Navigation

Figures 8-7 through 8-9 show UML class diagrams of the implementation of Argo/UML's views. The main window is implemented by class ProjectBrowser which consists of four panes: the navigator pane which shows navigational perspectives, the editor pane which shows diagrams and table views, the details pane which contains several tabs showing details of the selected design element, and the "to do" pane which presents feedback from critics using a dynamic "to do" list metaphor. The navigation pane, "to do" pane, and table views are discussed below. Secondary windows, such as the search window, are accessed through menu items in the menus of the main window.

The four main panes are always present in Argo/UML. The specific tabs shown in the editor pane and the details pane, however, are determined at system start-up time by a configuration file. Class ConfigLoader parses the configuration file and loads the classes that implement the various tabs. Each line of the configuration file contains a list of

alternative tab classes, which ConfigLoader attempts to instantiate. Using a configuration file allows new tabs to be added or existing tabs to be removed without recompiling Argo/UML.

Providing alternative implementations of some tabs allows Argo/UML to run in a somewhat degraded mode if requested library classes are not available. For example, the "Source" tab displays the Java source code that will be generated for the selected model. If Argo/UML is running in the JDK (Java Development Kit) environment, the source code will be colorized (keywords, strings, and comments are shown in different colors), but if Argo/UML is running in the JRE (Java Runtime Environment), the source code will be shown in black text only. In the example, the degraded mode is needed because the JRE does not provide the Java parsing library classes needed for colorization.

8.3.3 Implementation of Navigational Perspectives

Navigational perspectives are implemented as combinations of child generation rules. Argo/UML uses the Swing user interface library, which defines a TreeModel interface for use with its tree widget. Each TreeModel object implements methods to access or compute the children of a given tree node. Argo/UML adds a TreeModelComposite class to combine TreeModels and defines a set of approximately thirty traversal rules, each of which is itself a simple TreeModel.

Each of the traversal rules also includes two methods used to check the composition of navigational perspectives: prerequisite and provided. The prerequisite method returns a set of design element types, one of which must already be present in the set of types that the perspective can generate. The provided method returns a set of design element types

that can be generated by the rule. For example, the “Class->Initial States” rule has Class in its prerequisite set and State in its provided set. All navigational perspectives start at the object representing the entire design project and include Project in the set of design element types that can be reached. The addition of each rule adds to the set of reachable types. Rules with prerequisites that do not intersect the set of reachable types cannot be legally added and cause the rule addition button in the configuration window to be disabled.

Argo/UML uses standard JavaBeans event notifications to keep the navigator pane up-to-date when the design changes. As the designer expands or collapses the tree, the navigator pane adds or removes itself from the listener sets of the newly displayed or hidden design elements. When a design element changes state, it sends an event that causes the navigator pane to recompute the children of that element according to the current navigational perspective and to update the screen. This algorithm can be made scalable because it only needs to expend effort on tree nodes that are visible on the screen, regardless of the size of the design, the number of possible design perspectives, or the depth of the tree.

8.3.4 Implementation of the Dynamic “To Do” List and Clarifiers

Figure 8-7 shows the classes that implement the dynamic “to do” list. The “to do” list perspectives are implemented in much the same way as the navigational perspectives. However, the “to do” list perspectives are optimized for addition or deletion of items in batches, because the critiquing and item resolution threads work in cycles that produce batches of “to do” items.

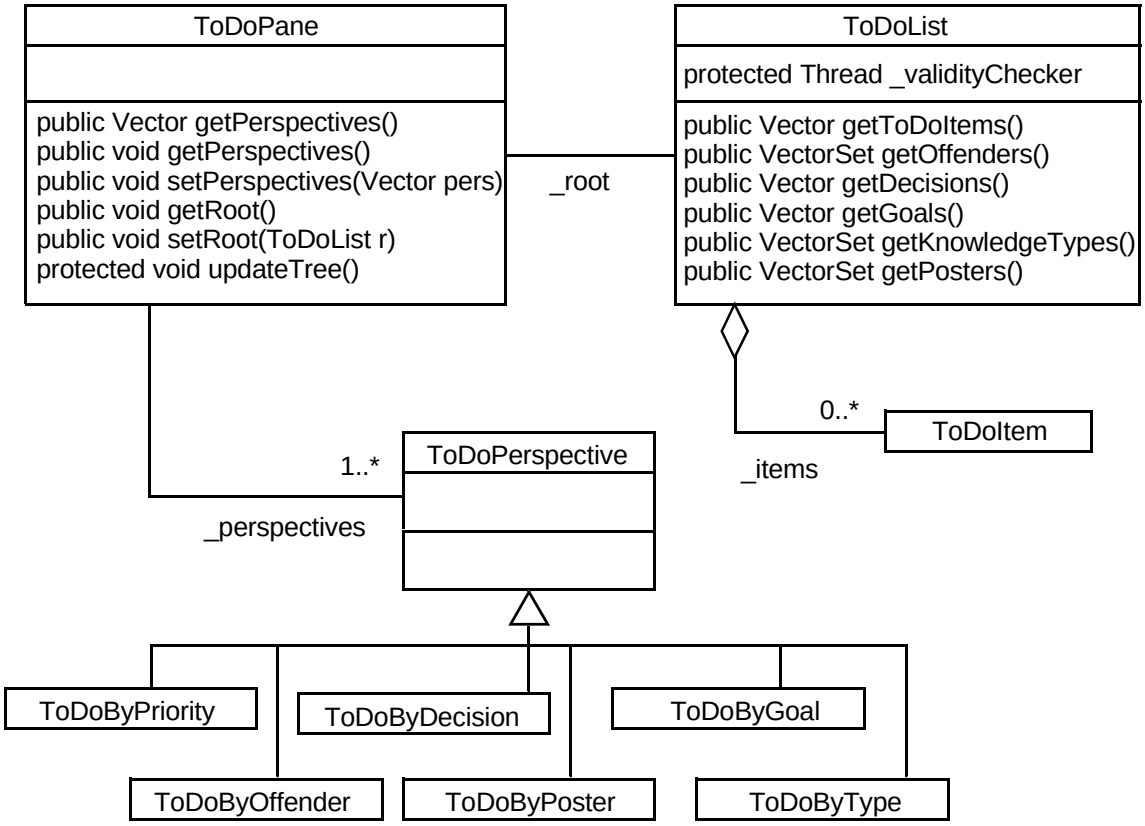

Figure 8-7. Classes implementing Argo/UML's "to do" list

Clarifiers are associated with the critics that produce each feedback item. The set of clarifiers to be displayed on a selected design element is computed simply by scanning the "to do" list for items that apply to the element, and drawing the clarifiers of the associated critics.

8.3.5 Implementation of Opportunistic Table Views

Argo/UML's table views are implemented by the classes shown in Figure 8-8. TablePanel is a base class for specific table views. It provides features common to all table views, including the labels and widgets at the top of the view that display the name of the view, show the number of rows, and allow the designer to change table perspective

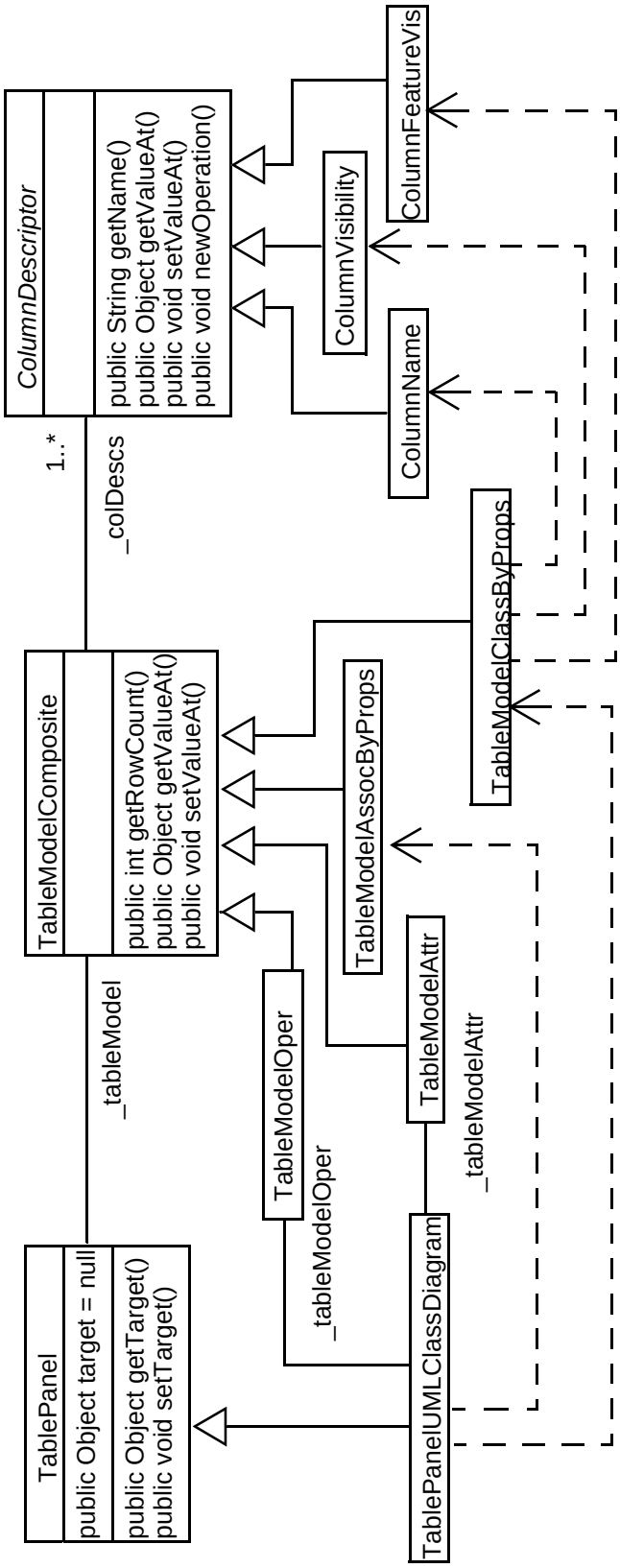

Figure 8-8. Classes that implement Argo/UML's table views

or configure the table or row filter.[3] JSortedTable is a specialized version of the standard Swing table widget that allows the user to sort the table by clicking on a column heading. Specific table views (e.g., TablePanelUMLClassDiagram) define the set of table perspectives (e.g., classes as rows or associations as rows) and any secondary tables (e.g., the tables of attributes and associations for the selected class).

Table perspectives are implemented as subclasses of class TableModelComposite, which implements the Swing TableModel interface by combining several column descriptors. For example, TableModelStateByProps consists of column descriptors for the name of the state, its entry and exit actions, its parent state, and its stereotype. ColumnDescriptor is an abstract base class for specific column descriptors. It keeps track of the column name, the type of value that will be displayed, and whether the table cells in that column should be editable. Specific column descriptors, such as ColumnName, implement accessor methods to get or set the appropriate value in a row object. Argo/UML currently provides thirty column descriptors to access the most commonly used model attributes; a set of approximately one hundred column descriptors would provide complete access to all of the attributes defined in the UML semantics specification (OMG, 1997).

8.3.6 Implementation of Opportunistic Search

Figure 8-9 shows the classes that implement Argo/UML's opportunistic search utility. Class FindDialog defines the layout of the widgets in the search window. Each widget in

3. Table configuration and row filtering are not implemented.

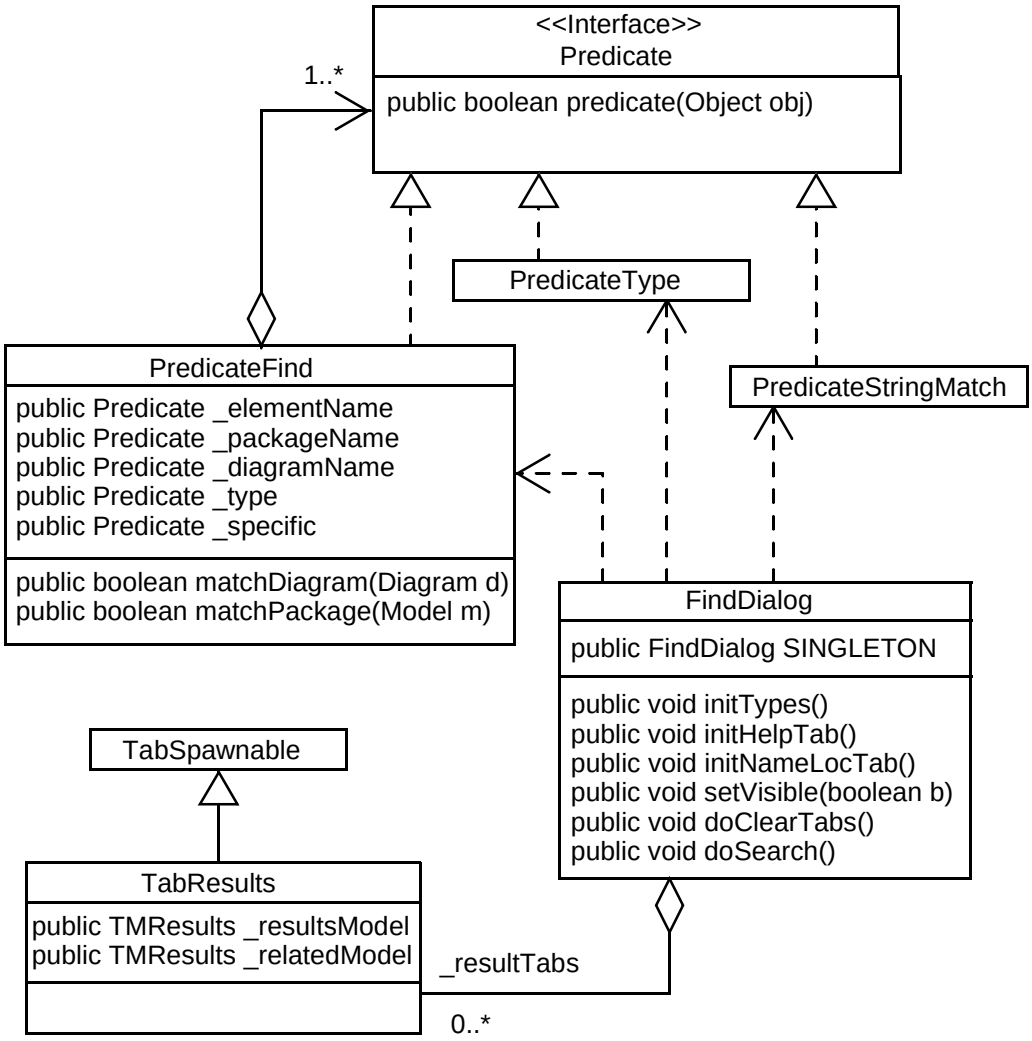

Figure 8-9. Classes implementing Argo/UML's opportunistic search utility

the top part of the window contributes a predicate object that is used to select search results. For example, the name field contributes a PredicateStringMatch object that selects only model elements with names that match the pattern entered in the name field. The individual predicates are combined into a PredicateFind object that performs a logical-and to select only those model elements that satisfy all predicates.

When the designer presses the "Search" button, a new TabResults is created to perform the search and display the results. The TabResults appears as a new tab in the lower half of the search window. TabResults defines two tables: one for the search results

and one for model elements related to the selected search result. The search result table is filled by traversing the design and applying the search predicate to each model element. The traversal is pruned if the designer specifies constraints on the packages or diagrams to be searched. The related elements table is filled by applying the rules in class ChildGenRelated. The rules in ChildGenRelated are currently implemented as Java code; a possible extension of this part of Argo/UML's infrastructure would allow designers to specify rules in a language such as OCL (Warmer and Kelppe, 1999).

8.4 Design Representation and Code Generation

8.4.1 Introduction

The preceding sections have described how Argo/UML presents design information to the user and provides knowledge support. This section covers how Argo/UML represents design elements internally and in external file formats.

8.4.2 Design Overview of Design Representation and Code Generation

Figures 8-10 through 8-12 show some of the classes that implement Argo/UML's design representation and the classes involved in processing model files and generating source files.

A recurring theme in this section of Argo/UML's implementation is the use of appropriate standards. UML (Unified Modeling Language) is a standard promoted by the Object Management Group (OMG, 1997). XML (Extensible Markup Language) is a standard for structured file formats promoted by the World-Wide-Web Consortium (W3C, 1998). XMI (XML Model Interchange format) is a standard way of storing UML designs

in XML files and is also promoted by the Object Management Group (OMG, 1998). PGML (Precision Graphics Markup Language) is a standard XML file format promoted by the World-Wide-Web Consortium for representing graphics that consist of primitive graphical elements such as lines, rectangles, and text (W3C, 1998). As will be further discussed in Chapter 9, leveraging standards guides development and reduces the need to develop and document new approaches.

8.4.3 Implementation of the UML Meta-Model

The UML standard consists of three main specifications: a notation guide that specifies the visual appearance of UML diagrams, a semantics specification that details the UML meta-model, and the OCL (Object Constraint Language) specification that adds a first-order predicate logic language for expressing constraints on UML models. The UML meta-model is itself a UML model that specifies how a UML design can be represented.

Figure 8-10 show some of the classes that implement Argo/UML's version of the UML meta-model. These classes were initially generated from a Rational Rose(tm) model provided with the UML 1.1 standard. As a result, Argo/UML strictly adheres to the UML standard, including all the names of packages, meta-classes, attributes, and associations. Leveraging the standard saved development resources that are very limited in an academic setting. Furthermore, strict adherence made it easier to support the XMI standard, which is itself generated from the UML standard.

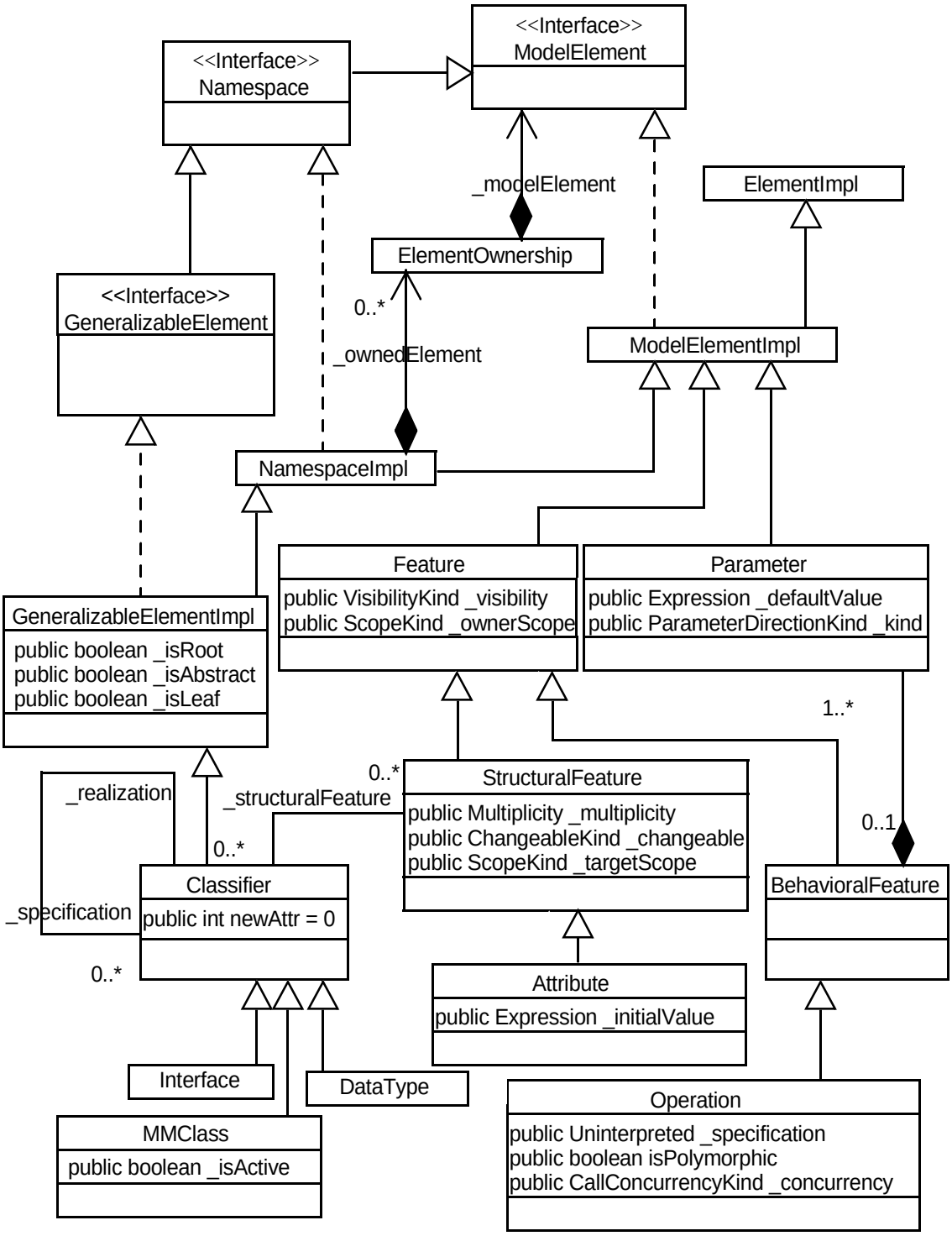

Figure 8-10. Some UML meta-model classes

Limited modifications were made to the meta-model to make it fit the Java language and the GEF library. For example, multiple inheritance used in the standard meta-model

was replaced with Java interfaces and single inheritance; also, an assumption in GEF required the addition of a Realization meta-class that is analogous to the Generalization meta-class. Fortuitously, recent changes to the UML standard match Argo/UML better than the earlier version (OMG, 1999).

Argo/UML's implementation of the UML meta-model uses JavaBeans-style method naming and change notifications. For example, the attribute "concurrency" of meta-class Operation in the UML meta-model is accessed with methods getConcurrency() and setConcurrency() in the Argo/UML implementation. Also, whenever the concurrency of an operation is changed, a standard JavaBeans property change event is fired with information about the name of the property that changed, its old value, and its new value.

Argo/UML's implementation of the UML meta-model consists of 9900 lines of Java source code in 103 classes. Test cases based on the examples in the UML specification add another 1800 lines in 15 classes.

8.4.4 Implementation of XMI and PGML File Formats

Argo/UML uses XMI files to store design representations. Using the XMI standard has helped keep the focus of Argo/UML on cognitive issues by allowing issues of interoperability, version control, and repositories to be deferred. Argo/UML uses IBM's XML parser to read XMI files using a straightforward set of tag handlers. It generates XMI files using a new "little language" called 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 evaluated with respect to a design element and

results in a bag of objects. Each result object is output in sequence and may use its own template. Table 8-2 gives two simplified examples of the TEE templates used to generate XMI.

Table 8-2: Some TEE templates for generating XMI files

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>
```

I chose not to use XMI's ViewElement, presentation, geometry, and style tags to represent diagrams. Instead, Argo/UML uses the PGML (Precision Graphics Markup Language) standard file format for diagrams (W3C, 1998). This has the advantage of being better defined and may allow users to view Argo/UML diagrams in future web browsers.

The UML standard is still evolving and the XMI standard is evolving with it. The UML 1.4 specification will be released next year with a new draft of the XMI

specification. Further revisions to UML are scheduled over the next few years. PGML has also evolved and has now been superseded by SVG (Scalable Vector Graphics) (W3C, 1999). The template expansion technique and TEE files are expected to make upgrading Argo/UML's file generation capabilities fairly easy.

The TEE file format and the use of embedded OCL expressions has proven remarkably flexible and useful, despite the fact that only a small subset of OCL is supported. One possible extension to the Argo/UML system would be to generate HTML reports to document designs on the web. Another extension would be to replace the code generation scheme discussed below with a more customizable one. Both of these can be accomplished with TEE files.

Figure 8-11 shows the classes that implement Argo/UML's XML file parsing. Altogether, Argo/UML's code to parse and generate XMI and PGML files consists of 4000 lines of Java code in 8 classes, 3 XML DTD (Document Type Definition) files totaling 3900 lines, and 3 TEE files totaling 2000 lines.

8.4.5 Implementation of Code Generation

Figure 8-12 shows the Argo/UML classes involved in code generation. Code generation in Argo/UML is supported with a language independent abstract base class and Java-specific subclasses. Class Generator is an abstract base class that is similar to the Visitor design pattern (Gamma et al., 1995). However, the logic to traverse the design representation is intermixed with node processing logic. Class GeneratorJava is a Java-specific subclass that generates Java source files. Class GeneratorDisplay generates

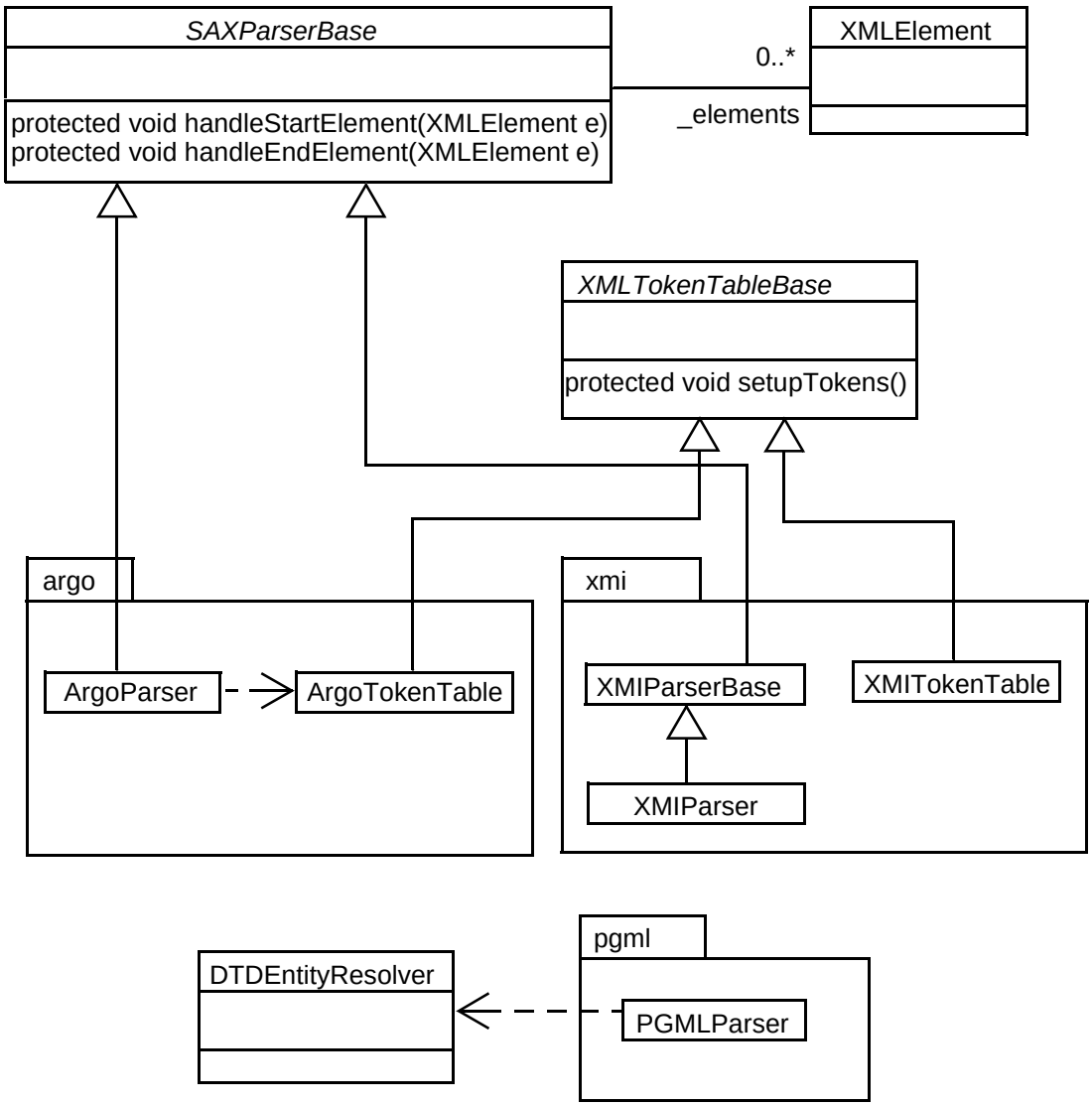

Figure 8-11. Classes implementing XML file processing

simplified Java code to be displayed in the “Source” tab and in the textual labels of UML class icons and other parts of UML diagrams.

Each of the Java-specific classes implements methods that generate source code for design elements of a given type. Since the code generation logic is coded in Java, the only way to customize it is by changing the code or by adding new code generation preferences. One possible extension to the Argo/UML system would be to use TEE files

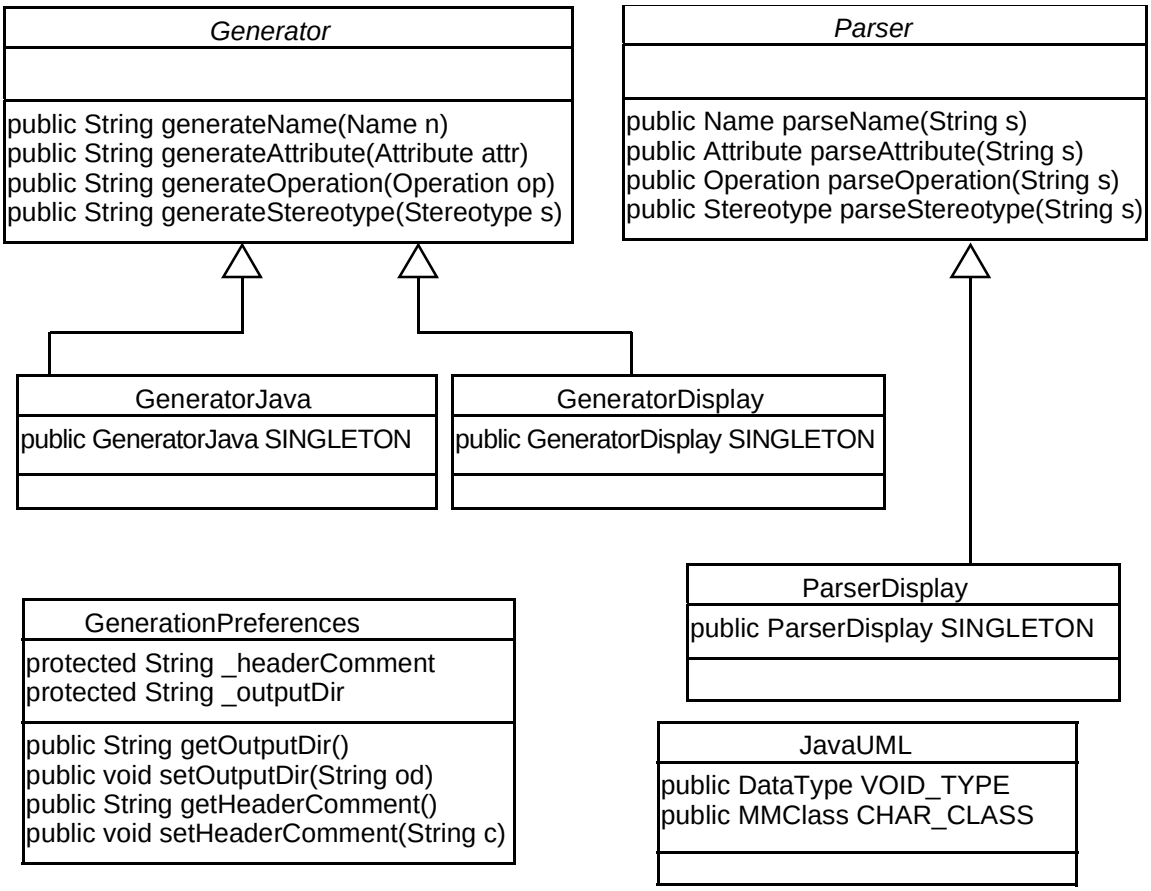

Figure 8-12. UML class diagram of classes for code generation.

to generate source code from templates, as is done for XMI and PGML files. This would greatly ease simple customizations. For example, with templates one can easily control the indentation of the generated code or the position of opening and closing braces. Furthermore, easily customized code generation might be useful in generating code other than the classes directly modeled. For example, a designer could generate property sheets by using a set of templates that generate one user interface window for each class in a design and one widget in that window for each attribute of the corresponding class.

Altogether, Argo/UML's code generation facilities consist of 2000 lines of Java code in 7 classes.

CHAPTER 9: Conclusion

This chapter reflects on the feature generation approach described in the dissertation, reviews my contributions, and outlines possible research extensions.

9.1 Reflections on the Approach

Design is a cognitively challenging task and designers can benefit from tools that support their cognitive needs. There is a substantial body of theory that describes these cognitive needs in the cognitive science literature. These theories are the result of many direct observations and laboratory studies. As I learned more about these theories, I became more interested in their underlying assumptions and practical implications.

The strong internal validity of these theories makes them good inputs to my feature generation approach. However, each theory only addresses one aspect of the overall cognitive challenge that designers face. Relying on any one theory alone can leave other cognitive needs unaddressed. So, multiple theories must be taken into account when designing a new feature. For a given feature, some theories provide positive guidance as to what the feature should *do*, while others provide negative guidance as to what the feature should *not do*. Furthermore, the cognitive theories do not specifically deal with user interface issues, so standard usability guidelines are also needed.

My feature generation approach has risks. Specifically, it requires practicing tool builders to gain a partial understanding of cognitive theories and then apply that knowledge. Since CASE tool builders typically do not have a background in cognitive

psychology, there is a significant chance of misunderstanding. Three aspects of the approach help mitigate this risk. First, supporting multiple theories helps to avoid the risk of relying too strongly on an incorrect, misunderstood, or misapplied cognitive theory. Second, evaluation techniques such as cognitive walkthroughs and user studies can find usability problems early. Third, cognitive support features are intended to support the designer in his or her design tasks; they should not interfere with the normal usage of the design tool and any potential interference is evaluated very closely.

The central and most difficult part of my feature generation approach is the invention of new features. How does one span the gap from theories to features? Although the gap may seem large, I have found the approach to be very productive. In some cases, features have been directly inspired by the theories; in other cases, the theories have merely guided feature development by confirming the value of intermediate development steps.

The opportunistic search utility and selection-action buttons are two examples of features directly inspired by cognitive theories. One interpretation of the theory of opportunistic design is that opportunistic task switching is normal and should be encouraged where appropriate. So, I reconsidered standard CASE tool features with the intent of providing additional information that might prompt opportunistic task switching while still aiding designers in returning from these design excursions. The result was Argo/UML's opportunistic search utility. Selection-action buttons were directly inspired by Fitts' Law and by the theory that limited short-term memory is used both for domain knowledge and tool interaction planning. The position of the selection-action buttons resulted directly from my understanding of Fitts' Law. The three modes of interaction

with selection-action buttons resulted from my efforts to match user interface affordances to common design tasks.

The development of the broom alignment tool is an example of using a theory to guide feature development. Once I learned about secondary notation, I knew that I wanted a feature to support it, but I did not have any specific ideas for that feature. One day, on a TV cooking show, the chef used a knife to chop vegetables and move the small pieces into distinct areas of the cutting board. That gave me the idea for the push-into-alignment action of the broom and emphasized the importance of tight integration into the normal diagram editing user interface. From there, the theory helped me choose which visual properties were most important to secondary notation, and usability guidelines suggested the broom's direct and reversible action.

9.2 Review of Contributions

The four contributions of this dissertation are the following:

- I described theories of design cognition in terms understandable and relevant to CASE tool builders. One of the reasons that current CASE tools do not satisfy their users is that CASE tool builders do not have a clear understanding of the user interface requirements of such tools. By describing the relevant cognitive theories in practical terms, I have helped clarify these requirements for myself and for other tool builders. These requirements inspired several novel features.

- I proposed a basket of useful CASE tool features. Each feature in Chapter 4 was inspired and explained by cognitive theories and HCI guidelines. Several of these features have intrinsic appeal and many evaluated well in Chapters 6 and 7. My basket of features is, in itself, a contribution to the CASE tool research area. In fact, some features have already been recognized by CASE tool vendors and are starting to appear in commercial products. In addition to the immediate contribution of these features, they also serve as examples that bring out key aspects of my feature generation research approach.

- I demonstrated the successful application of a theory-based user interface design approach to a large-scale software engineering tool. In a sense, every user interface is based on some tacit understanding of the user's cognitive needs. Some researchers have proposed tools with features based on a single cognitive theory (e.g., Guindon, 1992). However, relying on tacit understanding or individual theories of designers' cognitive needs has not yielded very useful and usable CASE tools. I have taken into account several complementary cognitive theories and user interface guidelines. The contribution of Argo/UML as a whole is the demonstration that theory-based user interface approaches can be scaled up to practical tools serving real designers, rather than research prototypes.

- I described the design of a scalable, reusable infrastructure for building the proposed features in design tools. Two of the reasons that design critiquing and other design support systems have never been widely adopted are (1) the lack of an infrastructure that can scale up to offer significant knowledge support, and (2) the lack of a reusable infrastructure that can compound progress made by different researchers. My critiquing infrastructure achieves scalability and reusability with a critic scheduling algorithm that provides good interactive performance while allowing critic authors to use the full power of the Java programming language.

9.3 Potential Extensions

The work leading to this dissertation has generated many interesting and promising ideas. Many of these ideas are visible in the Argo/UML tool. Others are described in the appropriate "Potential Extensions" sections of this dissertation. Still more ideas extend beyond what I have done. Four classes of possible extensions are discussed below:

- The first class of potential extensions continues along the same lines as my work to realize more of its potential benefits. One of the contributions of this dissertation is the demonstration of the productivity of my feature generation approach. Significant, immediate benefits have already been realized from this approach. Yet, the feature generation approach could be productively continued by adding new theories and features. Additional cognitive theories could be added to the body of theories considered. New features could be devised, current features could be refined, and features that are currently mock-ups could be fully implemented. Many of the proposed features could be evaluated in more depth and their heuristic evaluations can be further confirmed. Development of Argo/UML and user support could continue and the user population should continue to expand. These activities complement each other: additional theories inspire new features, new features require further evaluations, evaluation motivates the search for additional theories, further support for the design task would make Argo/UML more attractive to users, and more users would

provide more feedback that can provide additional confirmation of heuristic evaluations. Integration with the Expectation Driven Event Monitoring (EDEM) system (Hilbert and Redmiles, 1999) can also greatly increase the amount and usefulness of feedback from users.

- The second class of potential extensions would refine my reusable infrastructure to make it even more efficient, flexible, scalable, and understandable. Many of the trade-offs I have made in developing GEF, the Argo critiquing framework, and Argo/UML have emphasized simplicity and understandability. Only the critic scheduling algorithm and navigational perspectives have been optimized for efficiency. One of the simplifying assumptions of Argo/UML is that the entire system runs on a single Java virtual machine. This assumption could instead be relaxed to allow some critics to run on a remote Java virtual machine or be implemented in another language. Supporting critics in another language, such as C++, would require a translation of the Java critiquing framework into that language. Distributing the analysis load could scale up the amount of analysis carried out within the interactive time limit. However, a more significant benefit of distributed critiquing might be the ability to host critics on servers where they can be updated more easily by their authors.

- The third class of potential extensions focuses on applying my feature generation method and infrastructure to other design domains. Some of my proposed features have already been applied in four software tools that address very different aspects of software development. One such tool is Prefer, a state-based requirements modeling tool that includes design critics and the dynamic “to do” list. I also see a clear match between some of the cognitive support features and the cognitive challenges of a new task-based user interface design method (Constantine and Lockwood, 1999). Furthermore, designers in non-software domains face some of the same cognitive challenges. In fact, the cognitive theories are derived from observations of design and problem solving activity in many domains. For example, a specialized word processor could include critics and views that aid the writer in successfully completing a structured document such as the script of a play, a conference paper in a given field, certain types of grant proposals, or even a dissertation.

- The fourth class of potential research extensions would bridge the gap between the cognitive needs of individual designers and the needs of the development organization. These extensions would be based on theories of organizational memory, knowledge management, and on longitudinal studies of Argo/UML users. One of the advantages of design critics as they are described in this dissertation is that they can provide feedback that references the organizational context. For example, critics contain the email addresses of the experts who authored them. However, there is a body of research on organizational issues that has not been considered in my work. Refining and realizing the organizational aspects of design critics and investigating the interaction between cognitive theories and organizational theories should inspire many new features and lead to significant benefits.

REFERENCES

1. Ackerman, M. Augmenting the organizational memory: a field study of answer garden. *Proceedings of the 1994 Conference on Computer Supported Cooperative Work*. October 1994. Chapel Hill NC, USA. pp. 243-252.

2. Alexander, C., Ishikawa, S, Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. *A Pattern Language*. Oxford University Press. New York, NY. 1977.

3. Anderson, N. H. A functional theory of cognition. Lawrence Erlbaum Associates, Inc. Mahwah, NJ. 1996.

4. Apple Computer. *Macintosh Human Interface Guidelines*. Addison-Wesley. Reading, MA. 1993.

5. Austin, J. Four easy block breakers. (exercises to conquer writer's block). *Writer's Digest*. vol. 74. no. 3. March, 1994. pp. 32-34.

6. Beck, K. and Johnson, R. Patterns generate architectures. *Proceedings European Conference on Object-Oriented Programming (ECOOP'94)*. Bologna, Italy. 1994.

7. Bell, B., Rieman, J., and Lewis, C. Usability testing of a graphical programming system: things that we missed in a programming walkthrough. *Proceedings Human-Factors in Computing Systems (CHI'91)*. 1991. pp. 7-12.

8. Bertolazzi, P., Di Battista, G., and Liotta, G. Parametric graph drawing. *IEEE Transactions on Software Engineering*. Aug. 1995. vol. 21. no.8. pp.662-73.

9. Bonnardel, N. and Sumner, T. Supporting evaluation in design: the impact of critiquing systems on designers of different skill levels. *Acta Phychologica*. vol. 91. 1996. pp. 221-244.

10. Booch, G. *Object-oriented design*. Benjamin/Commings. Redwood City, CA. 1991.

11. Booch, G., Rumbaugh, J., and Jacobson, I. *The unified modeling language user guide*. Addison-Wesley. Reading, MA. 1999.

12. Boucher, S. Are you holding you back? (banishing writer's block). *Writer's Digest.* vol. 75. no. 4. April, 1995. pp. 30-34.

13. Brooks, F. P. No silver bullet: essence and accidents of software engineering. *IEEE Computer.* vol. 20. no. 4. April 1987. pp. 10-19.

14. Brooks, R. E. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*. vol. 18. pp. 543-554. 1983.

15. Byrne, M. D. and Bovair, S. A working memory model of common procedural error. *Cognitive science*. vol. 21. no. 1 (1997). pp. 31-61.

16. Carroll, J., Aaronson, A. Learning by doing with simulated intelligent help. *Communications of the ACM*. vol. 31. no. 9 Sept. 1988. pp. 1064-1079.

17. Chun H. W. and Lai, E.M.-K. Intelligent critic system for architectural design. *IEEE Transactions on Knowledge and Data Engineering*. vol. 9. no. 4. July/ August 1997. pp. 625-639.

18. Coad, P., Lefebvre, E., and De Luca, J. *Java Modeling in Color with UML: Enterprise Components and Process*. Englewood Cliffs, NJ: Prentice Hall. 1999.

19. Cofer, C. N. *The structure of human memory.* W. H. Freeman and Company. San Francisco, CA. 1975.

20. Collins, A.M. and Loftus, E.F. A Spreading-Activation Theory of Semantic Processing. *Psychological Review.* vol. 82. no. 6. Nov. 1975. pp. 407-428.

21. Constantine, L. L. and Lockwood, L. A. D. *Software for Use*. Addison-Wesley. Reading, MA. 1999.

22. Curtis, B., Krasner, H., Iscoe, N. A field study of the software design process for large systems. *Communications of the ACM.* vol. 31. no. 11. Nov. 1988. pp. 1268-1287.

23. Cypher, A., editor. *Watch What I Do: Programming by Demonstration*. MIT Press. Cambridge, MA. 1993.

24. Dryer, D. C. Wizards, guides, and beyond: rational and empirical methods for selecting optional intelligent user interface agents. *Proceedings 1997 International Conference on Intelligent User Interfaces*. Orlando, FL. January 1997. pp. 265-268.

25. Eckstein, R., Loy, M., and Wood, D. *Java Swing*. O'Reilly and Associates. Sebastopol CA. 1998.

26. Ellis, H. C. and Hunt, R. R. *Fundamentals of cognitive psychology.* 5th ed. 1993. Brown & Benchmark/Wm. C. Brown Publishers. Madison, WI.

27. Ercegovac, M. D. and Lang, T. *Digital Systems and Hardware/Firmware Algorithms*. John Wiley and Sons. New York, NY. 1985.

28. Fagan, L. M., Shortliffe, E. H., and Buchanan, B. G. Computer-based medical decision making: from MYCIN to VM. *Automedica*. Feb. 1980. vol. 3. no.2. pp. 97-106.

29. Faulk, S., Braket, J., Ward, P., and Kirby, Jr., J. The CoRE method for real-time requirements. *IEEE Software*. vol. 9. no. 9. pp. 60-72.

30. Finke, R. A., Ward, T. B., and Smith, S. M. *Creative cognition: Theory, research, and applications*. MIT Press, Cambridge, MA. 1992.

31. Fischer, G. and Morch, A. I. Crack: a critiquing approach to cooperative kitchen design. *Proceeding of the International Conference on Intelligent Tutoring Systems*. 1988. pp. 176-185.

32. Fischer, G. Cognitive view of reuse and redesign. *IEEE Software*. July 1987.

33. Fischer, G. Human-computer interaction software: lessons learned, challenges ahead. *IEEE Software*. January 1989. pp. 44-52.

34. Fischer, G., Girgensohn, A., Nakakoji, K., and Redmiles, D. F. 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. June, 1992. pp. 511-522.

35. Fischer, G., Grudin, J., Lemke, A., McCall, R., Ostwald, J., Reeves, B., and Shipman, F. Supporting Indirect, Collaborative Design with Integrated Knowledge-Based Design Environments. *Human-Computer Interaction*. Special Issue on Computer Supported Cooperative Work. vol. 7. no. 3. pp. 281-314.

36. Fischer, G., Lemke, A. C., Mastaglio, T., and Morch, A. I. Critics: an emerging approach to knowledge-based human-computer interaction. *International Journal of Man-Machine Studies.* vol. 35. no. 5. Nov. 1991. pp. 695-721.

37. Fischer, G., Lemke, A. C., Mastaglio, T., and Morch, A. I. The role of critiquing in cooperative problem solving. *ACM Transactions on Information Systems*. vol. 9. no. 2. April 1991. pp. 123-151.

38. Fischer, G., Lemke, A. C., McCall, R., and Morch, A. I. Making argumentation serve design. *Human-Computer Interactions.* vol. 6. no. 3-4. 1991. pp. 393-419.

39. Fischer, G., McCall, R., Ostwald, J., Reeves, B., and Shipman, F. Seeding, evolutionary growth, and reseeding: incremental development of design environments. *Human Factors in Computing Systems, CHI'94 Conference Proceedings 1994*. Boston, MA. April, 1994. pp. 292-298.

40. Fischer, G., Nakakoji, K., and Ostwald, J. Supporting the evolution of design artifacts with representations of context and intent. In *Proceedings of DIS'95, Symposium on Designing Interactive Systems.* Ann Arbor, MI. October, 1995. pp. 7-15.

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

42. Fitts, P. M. The information capacity of the human motor system in controlling amplitude of movement. *J. Experimental Psychology.* vol. 47. pp. 381-391. 1954.

43. Foner, L. N. Yenta: a multi-agent, referral-based match-making system. Proceedings First International Conference on Autonomous Agents. pp. 301-307. 1997.

44. Fox, R. News track. *Communications of the ACM*. vol. 40. no. 5. May 1997. pp. 9-10.

45. 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.

46. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley. Reading, MA. 1995.

47. Gertner A. S. and Webber B. L., TraumaTIQ: on-line decision support for trauma management. *IEEE Intelligent Systems*. January/February 1998. pp. 32-39.

48. Girgensohn, A. *End-user modifiability in knowledge-based design environments.* Ph.D. Thesis. University of Colorado at Boulder. Dept. of Computer Science. June 1992.

49. Glass, R. L. Inspections - some surprising findings. *Communications of the ACM*. vol. 42. no. 4. April 1999. pp. 17-19.

50. Graf, W. H. and Neurohr, S. Constraint-based layout in visual program design. *Proceedings 11th IEEE International Symposium on Visual Languages*. Darmstadt, Germany. 1995. pp.116-117.

51. Green, T. R. N. and Petre, M. Usability analysis of visual programming environments: a "cognitive dimentions" framework. *Journal of Visual Languages and Computing*. vol. 7. no. 2. June 1996. pp. 131-174.

52. Grudin, J. Error patterns in skilled and novice transcription typing. In *Cognitive Aspects of Skilled Typewriting*, W. E. Cooper, Ed. Springer-Verlag, New York. 1983.

53. Guerlain, S., Smith, P. J., Obradovich, J., Smith, J. W., Rudmann, S., and Strohm, P. The antibody identification assistant (AIDA), an example of a cooperative computer support system. *In 1995 IEEE International Conference on Systems, Man and Cybernetics*. Vancouver, BC, Canada, 22-25 Oct. 1995. p. 1909-1914.

54. Guindon, R. Requirements and design of DesignVision, an object-oriented graphical interface to an intelligent software design assistant. *Proceedings Human Factors in Computing Systems (CHI'92)*. 1992.

55. Guindon, R., Krasner, H., and Curtis, W. 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*. Ablex Publishing Corporation. Norwood, NJ. 1987. pp. 65-82.

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

57. Henninger, K. Specifying software requirements for complex systems: new techniques and their application. *IEEE Transactions on Software Engineering*. vol. 6. no. 1. January 1980. pp. 2-13.

58. Hilbert, D. M., Robbins, J. E., and Redmiles, D. F. EDEM: intelligent agents for collecting usage data and increasing user Involvement in development. *Proceedings of the 1998 International Conference on Intelligent User Interfaces (IUI'98).* San Francisco, CA. Jan. 1998. pp. 73-76.

59. Huff, C. C. Elements of a realistic CASE tool adoption budget. *Communications of the ACM*. vol. 35. no. 4. 1992. pp. 45-54.

60. Iivari, J. Why are CASE tools not used. *Communications of the ACM.* vol. 30. no. 10. Oct. 1996. pp. 94-103.

61. International Data Corporation (IDC). Object tools: 1996 worldwide markets and trends. International Data Corporation. 1996.

62. Jansson, D. G. and Smith, S. M. Design fixation. *Design Studies.* vol. 12. pp. 3-11. 1991.

63. Keller, R. K., Schauer, R., Robitaille, S., and Page, P. Pattern-Based Reverse-Engineering of Design Components. *Proceedings of the 1999 International Conference on Software Engineering*. Los Angeles, CA. May 1998. pp. 226-235.

64. Kintsch, W. and Greeno, J. G. Understanding and solving word arithmetic problems. *Psychological Review.* vol. 92. 1995. pp. 109-129.

65. Kintsch, W. and Polson, P. G. On nominal andfunctional serial position curves: implications for short-term memory models? *Psychological Review.* vol. 86. no. 4. July 1979. pp. 407-413.

66. Koenemann, J. and Robertson, S. P. Expert Problem Solving Strategies for Program Comprehension. *Proceedings Human-Factors in Computing Systems (CHI'91)*. 1991. pp. 125-130.

67. Krasner, G. E. and Pope, S. T. A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*. vol.1. no. 3. 1988.

68. Krueger, C. W. Software reuse. *ACM Computing Surveys*. vol. 24. no. 2. 1992.

69. 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.

70. Lee, J. Design rationale systems: understanding the issues. *IEEE Expert*. 1997. 78-85.

71. Lee, S. D. Toward the efficient implementation of expert systems in Ada. *Proceedings of the conference on TRI-ADA '90*. 1990. pp. 571-580.

72. Lemke, A. C., Fischer, G. A cooperative problem solving system for user interface design. *AAAI-90*. 1990. pp. 219-240.

73. Meyer, A. S. and Bock, K. The tip-of-the-tongue phenomenon: Blocking or partial activation? *Memory & Cognition.* vol. 20. no. 6. Nov. 1992. pp. 715-726.

74. Miller, G. A. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review.* vol. 63. no. 2. March 1965. pp. 81-97.

75. Miller, P. L. ATTENDING: critiquing a physician's management plan. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Sept. 1983, vol. PAMI-5. no. 5. pp. 449-461.

76. Nielson, J. *Usability Engineering*. Academic Press. Boston, MA. 1993.

77. Object Management Group (OMG). UML Specification v1.3: Object Management Group document ad/99-06-08. June 1999. Available from http://www.omg.org.

78. Object Management Group (OMG). XML Metadata Interchange (XMI): Object Management Group document ad/98-07-01. July 1998. Available from http://www.omg.org.

79. OMG (Object Management Group). UML Semantics. Object Management Group document ad/97-08-05. Sept. 1997. Available from http://www.omg.org/.

80. Osborn, A. *Applied Imagination*. 1953. Charles Scribner's Sons. New York, NY.

81. Petre, M. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*. June 1995. vol. 38. no. 6. pp. 33-44.

82. PITAC (The President's Information Technology Advisory Committee). Available from http://www.ccic.gov/ac/. 1997.

83. Porter, A. A. and Johnson, P. M. Assessing software review meetings: results of a comparative analysis of two experimental studies. *IEEE Transactions on Software Engineering*. vol. 23. no. 3. March, 1997. pp. 129-145.

84. Pree, W. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley. Reading, MA. 1995.

85. Quantrani, T. *Visual Modeling with rational rose and UML*. Addison-Wesley. Reading, MA. 1998.

86. Raisamo, R. and Raiha, K-J. A new direct manipulation technique for aligning objects in drawing programs. *Proceedings ACM Symposium on User Interface Software and Technology (UIST'96)*. 1996. pp. 157-164.

87. Raisamo, R. An alternate way of drawing. *Proceedings Human-Factors in Computing Systems (CHI'99)*. 1999. pp. 175-182.

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

89. Rieman, J., Franzke, M., and Redmiles, D. Usability Evaluation with the Cognitive Walkthrough. *Proceedings Human-Factors in Computing Systems (CHI'95)*. 1995.

90. Riesbeck, C. K. and Dobson, W. Authorable critiquing for intelligent educational systems. *Proceedings of the 1998 International Conference on Intelligent User Interfaces*. San Francisco, CA. January 6-9, 1998. pp. 145-152.

91. Robbins, J. E., Hilbert, D. M., and Redmiles, D. F. Extending design environments to software architecture design. In *Proceedings of the 11th Knowledge-Based*

Software Engineering Conference. Syracuse, NY, USA, 25-28 Sept. 1996. pp. 63-72.

92. Robbins, J. E., Kantor, M., and Redmiles, D. F. Sweeping away disorder with the broom alignment tool. *Proceedings Human-Factors in Computing Systems (CHI'99)*. 1999. pp. 250-251.

93. Robbins, J. E., Morley, D. J., Redmiles, D. F., Filatov, V., and Kononov, D. Visual Language Features Supporting Human-Human and Human-Computer Communication. *IEEE Symposium on Visual Languages 1996 (VL'96)*. Boulder, CO. Sept. 1996. pp. 247-254.

94. Rogers, I. The use of an automatic "to do" list to guide structured interaction. *Proceedings Human-Factors in Computing Systems (CHI'95)*. Denver, CO. May, 1995. pp. 232-233.

95. Ross, B. H. and Bower, G. H. Comparisons of Models of Associative Recall. *Memory & Cognition*. vol. 9. no. 1. 1981. pp. 1-16.

96. Rosson, M. B., Kellogg, W., and Maass, S. The designer as user: building requirements for design tools from design practice. *Communications of the ACM*. vol. 31. no. 11. Nov. 1988. pp. 1288-1298.

97. Roth, E. M., Malin, J. T., and Schreckenghost, D. L. Paradigms for Intelligent Interface Design. In *Handbook of Human-Computer Interaction, 2nd ed*. Eds: Helander, Landauer, and Prabhu. Elsevier Science. 1997. pp. 1177-1201.

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

99. Ryall, K., Marks, J., and Shieber, S. An interactive constraint-based system for drawing graphs. *Proceedings ACM Symposium on User Interface Software and Technology.* 1997. pp. 97-104.

100. Sannella, M. Skyblue: a multi-way local propagation constraint solver for user interface construction. *Proceedings of the ACM symposium on User interface software and technology.* 1994. pp.137-146.

101. Schoen, D. Designing as reflective conversation with the materials of a design situation. *Knowledge-Based Systems*. 1992. vol. 5, no. 1. pp. 3-14.

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

103. Seemann, J. and von Gudenberg, J. W. Pattern-based design recovery of java software. *SIGSOFT '98*. Nov. 1998. pp. 10-16.

104. Shneiderman, B. *Designing the User Interface, Third Edition*. Addison-Wesley. Reading, MA. 1998.

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

106. Simon, H. A. *The Sciences of the Artificial, 3rd ed*. MIT Press. Cambridge MA. 1996.

107. Smith, S. M. The TOTimals method: effects of acquisition and retention factors on tip-of-the-tongue experiences. International Conference on Memory. Lancaster, England. 1991.

108. Smith, S.M. and Vela, E. Incubated reminiscence effects. *Memory & Cognition*. vol. 19. no. 2. March 1991. pp. 168-176.

109. Smith, S.M. Frustrated feelings of imminent recall: On the tip of the tongue. In *Metacognition: Knowing about knowing.* Eds: J. Metcalfe, A.P. Shimamura. MIT Press, Cambridge, MA. 1994. p. 27-45.

110. Smith, S.M., Ward, T.B., and Schumacher, J.S. Constraining effects of examples in a creative generations task. *Memory & Cognition*. vol. 21. no. 6. Nov. 1993. pp. 837-845.

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

112. Soloway, E. and Ehrlich, K. Emperical studies of programming knowledge. *IEEE Transactions on Software Engineering*. vol. 10. no. 5. Sept. 1984. pp. 595-609.

113. Stacy, W. and MacMillian, J. Cognitive bias in software engineering. *Communications of the ACM*. vol. 38. no. 6. June 1995. pp. 57-63.

114. Stallman, R. M. EMACS: the extensible, customizable, self-documenting display editor. MIT AI Memo 519A. June, 1979.

115. Subramanian, R. and Adam, N. R. The design and implementation of an expert object-oriented geographic information system. *Proceedings of the second international conference on Information and knowledge management*. 1993. pp. 537-546.

116. Sumner, T., Bonnardel, N., and Kallak, B. H. The cognitive ergonomics of knowledge-based design support systems. *Proceedings on Human Factors in Computing Systems (CHI'97)*. 1997. pp. 83-90.

117. Sun Microsystems. *Java Look and Feel Guidelines*. Addison-Wesley. Reading, MA. 1999.

118. Taylor, R. N., Medvidovic, N., Anderson, K., Whitehead, Jr., E. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A component and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*. vol.22. no.6. June 1996. pp.390-406.

119. Teach, R. L. and Shortliffe, E. H. An analysis of physician attitudes regarding computer-based clinical consultation systems. *Computers and Biomedical Research*. Dec. 1981. vol. 14. no. 6. pp. 542-558.

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

121. Vlissides, J. M. and 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.

122. Ward, T.B. Structured imagination: The role of category structure in exemplar generation. *Cognitive Psychology.* vol. 27. no. 1. Aug. 1994. pp. 1-40.

123. Warmer, J. B. and Kleppe, A. G. *The Object Constraint Language: Precise Modeling With UML*. 1999. Addison-Wesley. Reading MA.

124. Warton, C., Rieman, J, Lewis, C., and Polson, P. The cognitive walkthrough method: a practitioner's guide. In Nielsen, J. and Mack, R. (eds.), *Usability Inspection Methods*. John Wiley & Sons, Inc. New York. 1994.

125. Wirfs-Brock, R. J. and Johnson, R. E. Surveying current research in object-oriented design. *Communications of the ACM*. vol. 33. no. 9. Sept. 1990.

126. World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.1. W3C Recommendation 10-Feb-98. Available from http://www.w3.org.

127. World Wide Web Consortium (W3C). Precision Graphics Markup Language (PGML): World Wide Web Consortium Note 10-April-1998. Available from http://www.w3.org.

128. World Wide Web Consortium (W3C). Scalable Vector Graphics (SVG) 1.0 Specification: World Wide Web Consortium Working Draft 1999-07-30. Available from http://www.w3.org.